Vom Fachbereich für Mathematik und Informatik

der Technischen Universität Braunschweig

genehmigte Dissertation

zur Erlangung des Grades eines

Doktor-Ingenieurs (Dr.–Ing.)

von

Dipl.–Inform. Antonio José Grau Vázquez

Computer-Aided Validation of Formal Conceptual Models

23. März 2001

1. Referent: Prof. Dr. Hans-Dieter Ehrich

2. Referent: Prof. Dr. Isidro Ramos Salavert

Eingereicht am 18. September 2000

Abstract

Conceptual modelling is the process of the software life cycle concerned with the identification and specification of requirements for the system to be built. In the last years, the ever increasing demands for software correctness have encouraged the use of formal methods in the modelling process. The use of formal specification languages provides more precise and concise specifications, and a basis for formal verification. Nevertheless, there is still a need for techniques to support the validation of formal specifications against the informal user requirements. The importance of early requirements validation is widely accepted. It is a well-known fact that errors and misunderstandings introduced in the early phases of systems development are the most difficult and costly ones to correct, unless detected early. A limitation of formal specifications is that they cannot readily be understood by users unless they have been specially trained. However, user validation can be facilitated by exploiting the executable aspects of formal specification languages. Through specification animation, users can test and investigate the dynamic behaviour of the specification in several scenarios to see if it adequately captures their real needs.

This thesis presents a systematic approach and workbench environment to support the construction and validation through animation of TROLL specifications. TROLL is a formal object-oriented language designed for the analysis and design of distributed information systems. Our approach is an iterative requirements definition process consisting of the formal specification of requirements, the automatic transformation of the specification into an executable form, and the interactive animation of the executable version to validate user requirements. To provide objects with persistence in the animation environment, we analyse how the static structure of TROLL objects can be mapped into relational tables. In order to execute the specification, we analyse the operational meaning of state transitions in TROLL, determine an execution model, and describe the transformation of the specifications into C++ code. We present a prototype implementation of the workbench environment.

Zusammenfassung

Die konzeptionelle Modellierung ist die Phase im Softwareentwurf, die sich mit der Identifikation und der Spezifikation von Systemanforderungen befasst. Die in den letzten Jahren zunehmenden Forderungen nach korrekter und zuverlässiger Software haben die Verwendung von formalen Methoden im Modellierungsprozess gefördert. Formale Spezifikationssprachen ermöglichen präzisere und eindeutigere Spezifikationen und stellen somit die Basis für die formale Verifikation dar. Trotzdem werden Techniken zur Validierung von formalen Spezifikationen bezüglich der informellen Benutzeranforderungen weiterhin benötigt. Die Wichtigkeit der frühen Validierung von Anforderungen ist in der Fachwelt weitgehend akzeptiert. Der Aufwand und die Kosten für die Korrektur von Fehlern und Missverständnissen aus den frühen Entwurfsphasen wird immer größer je später diese entdeckt werden. Ein Nachteil von formalen Spezifikationen ist, dass sie für Benutzer ohne entsprechende Vorkenntnisse nicht leicht verständlich sind. Die Einbeziehung der Benutzer in den Validierungsprozess kann jedoch durch die Ausführung einer formalen Spezifikation vereinfacht werden. Mit Hilfe der Animation können Benutzer das dynamische Verhalten der Spezifikation in unterschiedlichen Szenarien untersuchen und dadurch ihre gewünschten Anforderungen überprüfen.
Diese Arbeit liefert einen systematischen Ansatz und eine Entwicklungsumgebung für die Konstruktion von TROLL-Spezifikationen und deren Validierung durch Animation. TROLL ist eine formale objektorientierte Sprache für die konzeptionelle Modellierung von verteilten Informationssystemen. Unser Ansatz basiert auf einem iterativen Prozess zur Anforderungsdefinition bestehend aus der formalen Spezifikation von Anforderungen, der automatischen Übersetzung der Spezifikation in eine ausführbare Form, und der interaktiven Animation um die Benutzeranforderungen zu validieren. Um die Objektzustände in der Animationsumgebung persistent zu halten, wird untersucht, wie die statische Struktur von TROLL-Objekten in relationale Tabellen umgesetzt werden kann. Um die Spezifikationen auszuführen, wird zunächst die operationale Bedeutung von TROLL-Zustandsübergängen analysiert, dann wird ein Ausführungsmodell festgelegt, und anschließend wird die Übersetzung von den Spezifikationen in C++ beschrieben. Wir zeigen eine prototypische Implementierung der Animationsumgebung.

Acknowledgments

I want to express my gratitude to my supervisor Prof. Hans-Dieter Ehrich for having accepted me into his working group and for his constant guidance and support during the course of this work. I also wish to thank my co-supervisor Prof. Isidro Ramos Salavert for his encouragement and valuable suggestions.

For the pleasant working atmosphere I thank all my former and current colleagues of the Information Systems Group: Peter Ahlbrecht, Gabi Becker-Würch, Jutta Bleiß, Grit Denker, Christiane Eberhardt-Herr, Silke Eckstein, Peter Hartel, Mojgan Kowsari, Juliana Küster Filipe, Thomas Mack, Karl Neumann and Ralf Pinger.

I am especially grateful to Juliana Küster Filipe who nicely accompanied me in this long and hard way of the thesis. *Muito obrigado por tua amizade e por todos os bons momentos que passamos juntos!*

Thomas Mack kindly helped me in the implementation of the TROLL workbench. Moreover, he taught me how to get along with system administrators. *Danke Thomas!*

Grit Denker and Mojgan Kowsari always found time for answering my endless TROLL questions. Thank you for your support and the nice moments.

Most of the TROLL workbench implementation was carried out by students doing their student and diploma theses. I, therefore, want to acknowledge Arnim Gerstenberger, Jörg Hartmann, Torsten Rütters, Torsten Schaper, Stefan Schulte and Stefan Voecks.

For the motivation and support during the writing time of the thesis I want to express my appreciation to Udo Mitzlaff, Inma Rodríguez Carrión, José Sánchez Bisquert (especially for the nice postcards from Valencia) and Sofía Valenzuela.

I also wish to thank my parents Antonio and Mari Luz and my sister Luz for their love, encouragement and understanding. I dedicate this thesis to them. *¡Gracias familia!*

Finally, I am indebted to the Spanish Ministry of Education and Culture and the German Research Council DFG for the financial support of this work.

Contents

List of Figures

Chapter 1

Introduction

This chapter presents the motivation, context, objectives and structure of the thesis.

1.1 Motivation

The importance of conceptual modelling in the development of complex software systems is widely accepted. Conceptual modelling is concerned with the identification and specification of requirements for the system to be built. As a result, the conceptual model constitutes a requirements specification document that clearly and precisely describes what a system is intended to do. In the last years, the ever increasing demands for software correctness have encouraged the use of formal methods in the modelling process. The precise syntax and semantics of formal specification languages allows developers to write more precise, concise and unambiguous specifications. Furthermore, formal methods provide a mathematical framework for logical reasoning about the specifications. A main activity in the conceptual modelling process is the validation of the model against the informal user requirements. Validating the specification is of paramount importance, because although an implementation may be proven to be correct with respect to the specification, this is no help at all if the specification does not reflect adequately the user's needs and requirements. Validation of user requirements has traditionally been concentrated on testing program code prior to system installation. However, experiences have shown that requirement errors detected at late stages of the development process lead to a dramatic increase

of the software costs. Although formal methods improve the correctness and reliability of the software product by proving that specifications are consistent or that an implementation corresponds to the specification, they cannot prove the correctness of the specification with respect to the requirements. The main problem of requirements validation is that it concerns the interface between formality and informality. The validation process can only increase the confidence that the specification represents a system which will meet the real needs of the user. Validation requires an active participation of users because they provided the original requirements. So users should be able to understand the specification in order to find out possible misconceptions. Because of the complex syntax and semantics, a limitation of formal specifications is that they cannot be readily understood by users unless they have been specially trained. A technique for facilitating the user participation in the validation process consists in exploiting the executability of formal specification languages. The execution of formal specifications is usually called *animation*. Through animation, users can observe, experiment and test the dynamic behaviour of the specification in different scenarios to see if it meets their real needs. The requirements validation problem and its improvement through animation of formal specifications are the main motivation for this work.

1.2 Context

The work reported in this thesis has been developed in the context of the formal object-oriented specification language TROLL. v. 3.0 [DH97, Har97a, GKK+98]. TROLL is a language designed for the analysis and design of distributed information systems. The roots of TROLL can be found in earlier work mainly devoted to semantic foundations of object oriented specifications [SSE87, SFSE88, EGS90, ES90, SJE92, EDS93, EJDS94, SHJE94]. These articles have been the starting point for the design of a series of specification languages based on the object paradigm. The language OBLOG was presented in [SSE87, CSS89, Esp93]. In the following years and based on OBLOG, the languages GNOME [SR94] and TROLL [JSHS91, JSHS96, HSJ+94, HKSH94] have been developed. The design of the third and current version of TROLL has been significantly influenced by experiences gained in an industrial project located at PTB (Physikalisch-Technischen Bundesanstalt, the German National Institute of Weights and Measures) in Braun-

schweig [Kow96, HDK+97, SK97, KG98]. Current research directions focus on foundations, language concepts, applications and tools. Regarding theoretical foundations, distributed logics [ECSD98, EC00], module theory [Küs00a, Küs00b] and model checking [EP00] are being addressed. Work towards extending TROLL by a module concept is under investigation [Eck98]. Besides further application projects in cooperation with PTB, TROLL is being applied in a project which aims at combining the TROLL and Petri nets approaches to software specification in a railway traffic control application [EG01]. The purpose of this thesis has been the study and development of tools supporting the modelling and animation of TROLL specifications.

Work reported in this thesis started in 1996 and has been mainly supported by a PhD grant from the Spanish Ministry of Education and Culture, and by the German Research Council DFG under the priority programme "Integration of Software Specification Techniques for Engineering Applications".

1.3 Objectives

The main objective of this thesis is to provide a systematic approach and toolset to support the construction and animation of TROLL specifications. To this end, we establish the bases necessary for the construction of an animation environment and develop a prototype of such environment. Main requirements of the environment are: support for edition, syntax and static semantics analysis, automatic transformation of the specification into an executable form, and animation of the executable version in a persistent user-friendly environment. Regarding these requirements, the concrete objectives of the work are as follows:

- Analysis and development of context-sensitive rules to be checked during the static semantics analysis of the specification.

- Analysis and development of mapping rules from the static structure of the specification into relational database schemas. The database serves as object repository in the animation environment.

- Analysis of the operational meaning of state transitions in TROLL and determination of an execution model.

- Analysis of the transformation of TROLL specifications into C++ code to be executed in the animator.

- Development of a prototype version of a workbench environment to support the construction and animation of TROLL specifications.

The aims of the work will be described further in the next chapter after introducing the thesis context.

1.4 Thesis Outline

The thesis is structured as follows:

Chapter 2 presents firstly the context of the work. After a brief introduction to the phases of the software life cycle, the chapter describes conceptual modelling in requirements engineering. Since TROLL, the language to deal with in this thesis, is formal and object-oriented, these features are especially treated. Validation of formal specifications using animation techniques is discussed. Next, the chapter situates the work developed in this thesis in the context previously presented. Finally, the chapter points out related work.

Chapter 3 presents the modelling with TROLL and its graphical part OMTROLL. The language concepts are introduced by example. The same example will be used in Chapter 7 for describing the tools contained in the TROLL workbench. The chapter concludes with a brief description of the formal semantics of TROLL.

The construction of an executable prototype from a TROLL specification is the subject of Chapters 4, 5, and 6. Similar to the traditional construction of compilers for programming languages, it will be presented in two main phases: analysis and synthesis. The former concerns with the required static analysis and parser tree generation. The latter concerns with the generation of code.

Chapter 4 describes the analysis phase of the construction of a TROLL animator. This phase serves for two purposes. On the one hand, several analyses assure the syntax and static semantics correctness of the specification. On the other hand, an intermediate representation is generated from the specification. The chapter describes the data structure which holds this representation and defines TROLL context-sensitive rules to be checked in the semantic analysis.

Chapter 5 analyses how the static structure of TROLL objects can be mapped into relational tables and presents a set of mapping rules. In the animation system, these rules are used for generating a relational database schema which holds the state of the objects created in the animator.

Chapter 6 analyses the behaviour of TROLL objects and presents an execution model for state transitions. The chapter also describes the implementation of the execution model and the translation of TROLL specifications into C++.

Chapter 7 presents the TROLL workbench, a collection of software tools supporting the modelling and validation of TROLL specifications. The workbench includes a projects management tool, graphical and textual editors, a syntax and static semantics checker, a HTML code generator for hypertext navigation through the specification components, a database schemas generator, a C++ code generator and an animator. The chapter describes the workbench architecture and the functionalities of each tool by example.

Chapter 8 sums up the main contributions of this thesis and suggests some directions for further work.

Appendix A shows the syntax of OMTROLL and TROLL. **Appendix B** contains the TROLL specification of the example used throughout this thesis. The structure of the abstract syntax graph generated from the specifications is presented in **Appendix C**. Finally, **Appendix D** describes the translation from TROLL into C++.

Chapter 2

Conceptual Modelling and Validation Support

This chapter introduces the context of the thesis as well as related work. After a brief introduction to the software development phases, we describe conceptual modelling in requirements engineering. We emphasise on the combined use of formal and object-oriented techniques in conceptual modelling. Next, we discuss the validation of conceptual models using animation techniques. Finally, we describe the aims of the thesis and present work developed in similar approaches.

2.1 Software Development

In the early days of the computing technology, few thought that software would have the relevance that it has reached nowadays. Software development, a term that first came into use around 1959 [Cer98], was associated in its origins to the computer programming activity, i. e. how to put a sequence of instructions together to get the computer to do a specific task. In this period the model used was the so-called *code and fix* approach: First, write the code and then fix it to eliminate errors. As software evolved from small programs to large complex systems, people realised that software development was not just "coding and fixing" and that a better understanding of the software nature was urgently required. The term of *software crisis* emerged in the middle 1960s. Facts as software did not meet users' requirements, was expensive, unreliable and not delivered on time were some of the arguments

used by people in industry as well as in academic circles to justify such crisis. In 1968 a NATO conference was held in Garmisch, Germany, with the provocative title of *Software Engineering*. This title was meant to imply that to overcome the *crisis* a new discipline of software engineering was necessary. Software development should be based on theoretical foundations, standards, tools, methodologies and techniques as found in the traditional branches of engineering. Ever since, a lot of effort has been done to give an engineering approach to the construction of software. Software production process models, quality assurance models, management techniques, new languages, methods and CASE tools have been developed and standardised. Whether the crisis still exists today and whether the software discipline has become an established engineering are still a matter of debate in the computer community [BK96, Gla98].

As in other engineering fields, the production process is in software engineering extensively studied. The software development process is also commonly called the *software life cycle*. Different software life cycle models have been proposed. These models are abstract descriptions of how software systems should be developed and consist of a series of phases starting when the system is conceived and ending when the system is no longer available for use. Regardless of the model being used, the software production process includes traditionally a definite series of phases[1]. These phases are inspired by the waterfall model [Roy70] and may be described as follows [GJM91, MR91]:

Requirements Phase

The purpose of this phase is to identify and document the requirements for the system to be built. This phase is concerned with *what* the system should do, not *how* to do it. Requirements fall generally into two categories: functional and non-functional. Functional requirements specify functions that the system must be capable of performing and are described by a mapping from inputs to outputs. Non-functional requirements are also called system constraints and restrict the possible solutions to be considered in the following development phases. The requirements phase is usually divided into two subphases: *requirements analysis* and *system specification*. Requirements analysis is concerned with the elicitation of the requirements and is an information-gathering exercise to find out the users' needs. System specification is concerned with documenting unambiguously the gathered information. The result

[1] or 'canonical stages' as they are called in [MR91]

of this phase is a document referred to as the requirements specification [IEE84]. The document describing the functional system requirements is usually called the conceptual model. Conceptual modelling will be discussed in the next section.

Design Phase
The goal of the design phase is to describe the architecture, components and interfaces of the software system. The design phase comprises a preliminary and a detailed design. The preliminary design, also called architectural design, decomposes the software system into modules, which may in turn be iteratively decomposed into smaller submodules. The detailed design describes data structures and algorithms for each module such that it is ready for coding.

Implementation Phase
This phase is concerned with the coding, testing and integration of the modules defined in the design phase. This phase is usually explicitly separated into two phases: *coding and module testing* and *integration and system testing*.

Delivery and Maintenance Phase
After the testing, the software system becomes operational and is delivered. The tasks of software maintenance are to detect and repair errors that occur after deployment and to carry out system modifications and extensions.

Structuring these phases, how they are related, who their participants are and which activities they comprise depend on the process model being used. The *waterfall model* assumes a sequential structure among phases. Although the waterfall approach has been widely used, it has also been enormously criticised. The assumption that software development may be carried out sequentially from requirements down to implementation and that each phase must be completed before the next can start is unrealistic. Another weakness of the waterfall model is that working software is not available until the end of the development cycle, thus feedback from end users is only provided at a very late stage. The *evolutionary model* proposes an incremental approach where parts of some phases are postponed in order to produce results from other phases earlier. In this approach, feedback from the users is received by delivering prototypes of the system. The *transformation model* intends

to obtain the final system by applying a sequence of transformations to a formal specification. The *spiral model* proposed by Boehm [Boe88] is considered a metamodel which helps to choose the most appropriate development model for a given software situation. The *object-oriented model* applies the concepts of object technology, as found in object-oriented analysis, design and programming languages, to the software development life cycle. Other models and approaches have been proposed that are similar or extensions of the cited above. For a more detailed description of software development process models the reader is referred to [GJM91, MR91, TD97].

2.2 Conceptual Modelling

As mentioned in the last section, the requirements phase has the aims of precisely establishing and documenting the requirements that have to be fulfilled by the system. The achievement of these aims is of paramount importance for the success of the entire development process. As reported by Boehm [Boe81], the relative cost to find and fix an error grows exponentially the later it is found. An error made in the requirements definition may suppose an increment of the cost by a factor of 100 times if it is not found until after the product has been delivered. Requirements engineering is the field of software engineering concerned with the acquisition and formalisation of user requirements. The fact that requirements should be specified in a natural, formal and abstract way has encouraged the use of conceptual modelling languages to describe such requirements at a *conceptual level*. Following [RC92], the conceptual modelling process consists of four main activities:

- *Knowledge acquisition* aims at capturing knowledge about the users' needs. Techniques used for acquiring such knowledge are natural language as interviews, forms, and if possible, reuse of specifications and reverse engineering.

- *Conceptualisation* concerns with the specification of the functional requirements using a conceptual modelling language and incorporates both static and dynamic properties of the application domain.

- *Validation* has the objective of checking whether the conceptual model is consistent and whether it correctly and adequately expresses the requirements informally stated by the users. Model validation will be discussed in Sect. 2.3.

- *Evolution management* is concerned with the model's adaptation to changes occurred in the requirements.

The result of the conceptual modelling process is a conceptual model which constitutes a requirements specification of the system to be built. This specification is used for two purposes. Firstly, it must be validated by end users in order to ensure that it describes their needs. A requirements specification may also be regarded as an agreement or contract between developers and users. Secondly, it is used by the system designers to develop a solution that meets the requirements. According to [ISO82], the contents of a conceptual model should follow two major principles: the *100% principle* and the *conceptualisation principle*. The former means that the conceptual model must define *all* relevant static and dynamic aspects of the application domain. The latter means that the conceptual model must *only* include conceptual relevant aspects. Based on these principles, several desirable properties for a conceptual model have been proposed [IEE84, Rom85, Sto91, Lou92]:

- *Implementation Independent*: A conceptual model should not include implementation details. It should be specified at a high level of abstraction and not lead developers to specific design or implementation solutions.

- *Unambiguous*: Every requirement expressed in the model should have a single interpretation.

- *Complete*: All relevant aspects of the system should be described in the specification.

- *Consistent*: A model is consistent if it does not include parts that are in contradiction.

- *Analysable*: It should be possible to carry out completeness and consistency checks on the model. A model should be *validatable*. This implies that users should be able to read and understand the model in order to see if it meets their needs. It should be also possible to *verify* that the system design and implementation satisfy the original requirements. Moreover, in order to be able to use the model for testing the implementation, a model should be *testable*, i. e. it should be quantitative enough so that testing may take place.

- *Traceable*: This property refers to the ability to cross-reference aspects of the requirements specification with aspects of the design or implementation

- *Modifiable*: The evolutionary nature of user requirements demands that a conceptual model has to be easily modifiable and adaptable to changes.

Other properties may be derived from the above ones. On the one hand, user validation requires that a model should be easily *understandable* by users. On the other hand, lack of ambiguity, consistency, and verifiability require *formality*. The last two properties seem to conflict with each other. Indeed, the use of formal notations is usually a handicap for users in order to be able to understand a specification. Model *executability* refers to the ability of a specification to be simulated against user requirements and plays an important role in requirements validation if formal methods are used.

Conceptual modelling has been influenced by different areas of Computer Science. Within artificial intelligence, conceptual modelling has been studied in *semantic networks* as a technique of knowledge representation. The use of modelling techniques in programming languages was introduced by *Simula*. In the databases field, semantic data models have proposed powerful modelling techniques for dealing with data intensive systems. The *Structured Analysis and Design Technique* (SADT) contributed to the use of conceptual modelling for requirements modelling. For a survey and classification of conceptual models see for instance [Myl98]. Since the late 1980s the object-orientation approach is influencing significantly conceptual modelling with new concepts and techniques. In particular, the combination of object-orientation and formal methods is a promising area of research, which aims at obtaining benefits from both fields. The next subsections will introduce formal methods, object-oriented modelling techniques and the benefits of their integration.

2.2.1 Formal Methods in Conceptual Modelling

Modelling techniques may be classified depending on their level of formality. Requirements can be specified *informally* using natural language. Informal specifications present the same problems inherent to natural language: they are inconsistent, inaccurate and ambiguous. The use of *semiformal* notations such as diagrams and tables improves the accuracy of the models, but they

still lack of precise semantics and have generally a *free* interpretation. *Formal* methods aim at overcoming these limitations by introducing mathematical rigour to the process modelling. Formal specifications are written in a language with precise syntax and semantics. Formality allows not only to write more precise and unambiguous specifications but also allows mathematical reasoning about them. The principal advantages of using formal methods are [BMP92]:

- Formal specifications are more precise and less open to misunderstandings than specifications written in a natural language.
- Formal methods support the proof of properties of a specification detecting inconsistency or incompleteness.
- Formal specifications may be animated to provide a system prototype.
- Formal methods support formal verification, i. e. the construction of formal proofs that an implementation satisfies a specification.
- Formal specifications may be automatically transformed into programs.
- Formal specifications have lower maintenance costs than those written in a natural language.

In the last years, formal methods have grown in popularity. They constitute currently a great topic of research in the academic community and are also being used successfully in industrial projects, especially in safety critical and security systems. Nevertheless, formal methods are usually not well understood and some misconceptions or 'myths' about them still exist [Hal90, BH95]. Although formal methods improve the correctness and reliability of the software product, they do not cover all aspects of software quality as efficiency or user friendliness. Furthermore, formal methods do not guarantee the correctness of a specification with respect to the requirements. They can prove that a specification is consistent or that an implementation satisfies a specification, but they cannot prove that a specification describes correctly the informal user requirements. One handicap when using formal methods is that formal specifications are normally very difficult to understand for users not familiar with mathematical notations. Possible solutions for making a formal specification comprehensible to users are paraphrasing the specification in natural language or animating the specification. These

techniques will be discussed in Sect. 2.3. Correctness proofs present also some limitations. Since proofs are carried out on idealised abstract machines, proving that a program satisfies a specification does not assure that its execution on a physical machine will be correct. Besides physical limitations, such as memory boundaries or correct implementation of real numbers, it is possible that the compiler, the operative system, or the underlying hardware do not work properly. These limitations do not mean that formal methods are useless. Formal methods help considerably to reduce ambiguity in the specifications and to increase the confidence in the correctness of programs. A more detailed discussion about the advantages and limitations of formal methods can be found in [Kne97, Vie97, Sai97].

2.2.2 Object-Oriented Modelling

One main issue in conceptual modelling is that a model has to describe both structural and behavioural aspects of a system. Traditionally, modelling techniques have only focused on one of these aspects. In the 1970s structured methods focused on the function as the building block of a system. Structured methods helped developers to organise the software by functions, but not to manage data. Conversely, semantic data models provided developers with a new approach of developing software based on data items as the building blocks. Data modelling methods had the opposite weakness of structured methods. They helped developers to manage the data but not to manage the functions. Since the late 1980s, however, the object-oriented approach has contributed to integrate both aspects in only one formalism. In the object-oriented approach, the building block of a system is the object. Objects are autonomous atoms of computation containing both structure and behaviour. Object-oriented principles as object encapsulation, information hiding and abstraction allow software developers to manage in a structured and better way the complexity of the problem domain.

The object-oriented approach has influenced many areas of software engineering. In the area of programming languages, object concepts were first supported by *Simula 67* [DMN67] and gained wider acceptance with *Smalltalk* [GR83]. Since then, numerous object-oriented languages such as *Eiffel* [Mey92], *C++* [Str97] and *Java* [AJ96] have emerged. Some database systems such as O_2 [BDK92], *GemStone* [BOS91] and *Jasmine* [IYIK96] support the definition of object-oriented database schemas. A large variety of object-oriented analysis and design techniques have been developed such as

OMT [RBP+91], *Booch* [Boo93], *Syntropy* [CD94] and *Fusion* [CAB+94]. Recently, *UML* [BRJ98], a joint of several well-known object-oriented techniques has been accepted by the Object Management Group[2] as standard notation for object-oriented specifications and is also becoming an industry standard.

In the last years, the field of combined formal and object-oriented techniques has become an active area of research. This combination brings important benefits to both techniques. On the one hand, object-oriented techniques provide structuring and naturalness to formal techniques and therefore they make formal models easier to handle. On the other hand, formal techniques provide a sound formal basis to object-oriented techniques. In particular, the benefits that formal methods may obtain from the object-oriented approach are [RPS95, AB91]:

- *Naturalness:* The naturalness and simplicity of object-oriented concepts can make formal specifications languages more attractive and easy to use. The view of the world as a dynamic network of collaborating objects allows developers to capture the reality in a more intuitive and natural way.

- *Abstraction:* The large variety of abstraction mechanisms supported by the object-oriented paradigm such as classification, generalisation and aggregation improve considerably the structuring of models. Object encapsulation of data and operations and message-passing as the only possible mechanism to read and change an object's state help to clearly distinguish between *what* operations do and *how* they do it. Additionally, by using class specialisation, object-oriented specifications can be managed at different levels of abstraction. This helps to control the complexity of an application by defining properties at the most appropriate abstract level.

- *Concurrency:* The view of the world as a collection of independent objects working in parallel and communicating with each other provides an excellent and natural means to express concurrency.

[2]The Object Management Group (OMG) is the software industry's largest consortium that produces and maintains computer industry specifications for interoperable enterprise applications. Its web page is located at http://www.omg.org

- *Extensibility:* A class hierarchy is easily extensible. Class specialisation enables new classes to be defined from old ones. In this way, specifications can be gradually extended as our understanding of the system domain grows.

- *Reuse:* Common properties between classes can be reused through inheritance relationships, so they do not need to be defined in every class. Furthermore, specification reuse is not only possible within a single system, but reuse between different systems is also facilitated through shared object libraries.

- *Seamlessness:* The emergence of the object-oriented paradigm has led to a new software life cycle model based on the use of objects and classes through all software development phases. The object-oriented software life cycle presents a seamless integration of all phases improving considerably the consistency between them. By using the same principles, formal specifications can be better integrated in the object-oriented development cycle.

The benefits of using formal methods have been already discussed in the previous section. From an object-oriented perspective, formal techniques allow us to give a precise meaning to complex object-oriented mechanisms such as aggregation and subtyping and provide a rigorous basis for validation, verification and tool support.

In recent years, a large variety of specification languages which make use of both formal and object-oriented techniques have been developed. They are either new languages such as *OASIS* [PR95, LRSP98], *ALBERT* [DB95] and TROLL [GKK+98] or extensions to well-known languages such as *VDM++* [DK92], *Z++* [Lan91] and *LOTOS* [CRS90]. An interesting approach is the combination of semiformal and formal object-oriented techniques. On the one hand, semiformal object-oriented analysis and design techniques provide developers with a collection of diagrams that are intuitive and can be used as a means of communication between people not familiar with formal methods. On the other hand, formal techniques allow mathematical reasoning about the specification. The advantage of this combination is that the most appropriate notation may be used when modelling a system. Translation between graphical and formal notations is usually supported by software tools that ensure the consistency between them. Such is the case in the Venus toolkit [Ver96] combining OMT and VDM++ notations; OO-Method

[PPIG98] which translates UML compliant diagrams into OASIS, and the TROLL workbench [GK97], the work developed in this thesis, combining OMTROLL diagrams and TROLL.

2.3 Validation of Formal Specifications

Validation is the process of checking whether the specification captures the informal requirements as established by the users. Validation of user requirements is of an extremely importance in order to assure the reliability and correctness of the software product. Although one can verify the correctness of an implementation with respect to a specification, this is no help at all if the specification does not match the user requirements. Validation of user requirements has traditionally been concentrated on testing program code prior to system installation. However, it is a well-known fact that requirement errors detected at this stage are enormously expensive compared to earlier detection correction. It is thus important that specifications be validated prior to the implementation. The main problem of the validation process is that it concerns the interface between formality and informality and therefore it is not possible to *prove* that the specification is correct with respect to the requirements. Stokes ([Sto91]) uses Kant's distinction between analytical and synthetic reasoning to explain this fact. On the one hand, in analytical reasoning any proposition, i.e. any statement which can be reduced to either true or false, can be evaluated purely within some logical framework. For instance, verification of consistency between two formal abstract descriptions of the system is an analytical process. On the other hand, in synthetic reasoning a proposition can be judged to be true or false only by making an observation of the real world. Such observations are subjective, i. e. different persons may have different opinions, and therefore we cannot give an absolute truth value to any statements concerning the real world. Validation entails synthetic reasoning and therefore can only increase our confidence that the specification meets the real needs of the users, but not to prove it. Validation must involve users since they provide the original requirements. This requires that they understand the relevant parts of the specification and are able to communicate their subjective judgements on the basis of it. Formal specifications are precise, unambiguous and support formal verification of system properties by allowing mathematical reason-

ing[3] about them. However, their formal notations are usually difficult to understand by users. Several techniques have been developed for improving the users' understanding of the specifications and, hence, facilitate their participation in the validation process. They can be classified as follows [DDD94, Gul96]:

- *Conversion Techniques:* This group of techniques is concerned with the presentation of the models in a more easily understandable way. The introduction, for instance, of graphical symbols or user-defined concepts make specifications more intuitive and accessible to users. Another possibility consists in reducing the complexity of the models by suppressing irrelevant details and highlighting relevant ones [Sel93]. Another technique is to paraphrase specifications into natural language allowing users to read them in a language with which they are familiar [RP92, Dal92]. In this context, generation of abstractions and abstracts of specifications have also been investigated [JC92].

- *Behavioural Techniques:* These techniques emphasise on validating the dynamic properties of the model through its execution [Har92]. In this way, users can observe, experiment and test the dynamic properties of the model making easier its comprehension. The possibility of animating a prototype of the specification depends on the level of executability of the specification language. Animation will be presented in further detail in the next section.

- *Analysis Techniques:* These techniques aim at improving the validation process by analysing the model or its execution and presenting the results in an adequate form to the users. Coming from the expert systems research area, explanation generation techniques are concerned with the generation of explanations related to the modelling language, the model or its execution [Gul96, OS96]. Formal verification techniques can also be used for demonstrating consequences of the specification which are then presented, under a suitable interpretation, to the user for his/her approval [Hal90].

[3]The main approaches to verification are model-checking and theorem proving. Model-checking consists in building a finite model of a system and automatically checking that a desired property holds in the model. Theorem proving is the process of finding a proof of a property from the axioms of the system and may be partially automated. See, for instance, [CW96] and [BLR99] for an introduction to these techniques.

Since this thesis is concerned with providing animation support to TROLL, we describe animation techniques in more detail in the next section.

2.4 Validation through Animation

A major approach to the early validation of user requirements is based on exploiting the executability of formal specification languages. A conceptual model with executable properties can be interpreted as or automatically transformed to an executable prototype that can be evaluated to detect potential misconceptions expressed in the model. This process is usually called animation [HJS93, MJHB98]. Animation combines the advantages of formal specifications (e. g. unambiguity and analysability) and rapid systems prototyping (e. g. risk management and early user involvement). This is an attractive idea because the specification of a software system has to be produced, and no extra effort for prototype construction would therefore be needed. This is in contrast to other techniques, such as those using very high-level languages where, after the prototype has been built and agreed with the users, the specification has to be constructed anew. The fact that the prototype is directly derived from the specification assures the consistency between the specification and the observed behaviour of the prototype. Animation can be embedded in various software development paradigms. Especially attractive are the combinations with protoyping, and with the operational and transformational approaches [Agr86]. As indicated in Fig. 2.1, the software development model followed by animating formal specifications starts with a requirements analysis. Afterwards an iterative requirements definition process is carried out which consists of the formal specification of requirements, the construction of the animation and the validation and elicitation of new requirements by executing the animation. Once the formal requirements specification is considered satisfactory, the realisation process starts. This process can either consist of conventional design and implementation phases, or it can be based on transformational implementation, i. e. the iterative application of transformations on a formal specification, which leads to the final product. The latter is usually considered in the literature as transformational programming [Zav84, PS83]. The goal of transformational programming is to transform a specification automatically into an executable software system that is certain to satisfy the initial specification and in which validation and maintenance are done at the specification level [Bal85].

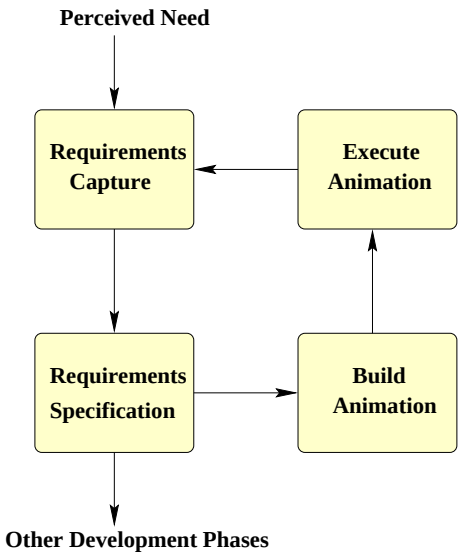


Figure 2.1: The Animation Process

The possibility of executing a specification depends on the degree of executability of the specification language. There is no consensus whether a specification should be executable or not. On the one hand, specifications should be declarative, i. e. they should describe *what* should be done and not *how* to do it. Hayes and Jones [HJ89] argue that executable specifications lack the expressibility of their non-executable counterparts, and that the demand of executability in the specification could influence implementation decisions. On the other hand, executability would be enormously profitable in order to be able to construct powerful validation tools. As argued by Fuchs [Fuc92], executable specifications allows us to obtain executable components at a very early stage, thereby allowing the earlier detection and correction of problems and the clarification of requirements that are unclear.

There are several ways of animating a specification [Har92]. The specification can be animated in an interactive fashion allowing an explorative validation of the user requirements. Users can emulate the system's environment by generating events. The animator, in turn, responds by transforming the system into the new resulting status. If the model is represented graphically, the change in status could be reflected visually, for instance, by changes in colour or emphasis in the diagrams [KK88, LL93]. Another possibility is

to execute the specification in a batch fashion. In this case, the sequence of events are read from a file. This allows outputs from scenario design editors [KSTM98] or specification-based test case generators [Pos96] to be used as inputs of the animator. Analogous to debugging tools of conventional programs, it is also possible to incorporate breakpoints, causing the execution to suspend and the animator to take certain actions when particular situations come up [Muk95]. For non-executable parts of the specification, it is sometimes possible to refine them using executable constructs of the specification language [Muk95, HJS93]. If such transformations are not possible, for instance, if the specification is non-deterministic or undetermined, the animator could warn or ask users to make a decision whenever they try to execute such expressions [Hey97]. Symbolic execution can also be used for animation purposes to specify symbolic values when the inputs, outputs or algorithms are not determined, or at least not uniquely [JJLM91].

An animation session would consist in identifying first a set of *scenarios*, where a scenario is a sequence of events which may occur in the domain of the system. This sequence of events is then tested against the specification to see whether the specification meets the intended user requirements involved in the scenario [Lan95]. Scenarios may represent both desired behaviour which the specification should allow, and undesired behaviour, which it should forbid. An animation session should be carried out by specifiers together with end users in order to observe and discuss whether the specification adequately describes the requirements that the future software system must fulfil. Users should bear in mind that the interface of the animation tool does not represent the eventual user interface of the system being developed. Since an exhaustive animation, i. e. testing the specification in all possible scenarios, is not practicable, a representative set of scenarios must be chosen. If requirements have been elicited using scenario-based techniques, scenarios obtained in the elicitation phase are the most appropriate to validate the specification [HD98, Sán00]. Another possibility consists in the use of automatic specification-based software testing techniques for obtaining test suites directly from the specification [Pos96]. The test suites which are used to test the specification can be reused later when testing the implementation. This helps to ensure a correct transformation from the specification into an implementation. Once the specification has been animated and observed that its behaviour meets the user requirements in a set of scenarios, we would like to affirm that such specification is *correct* with respect to the requirements. Unfortunately, a correct behaviour of the specification on a finite number of

cases does not guarantee correctness. Thus, as Dijkstra said in relation to program testing [DDH72] , we can say that animation can be used to show the presence of bugs, but never to show their absence.

Another issue is how to check whether the animation tool or prototype conforms to the dynamic semantics of the specification language. Several ways of doing this are discussed in [LP92]. As the compliance of compilers for programming languages is traditionally checked, one way is the use of test suites. The strategy consists in providing a significant number of tests, which the tool must execute in the way described by the semantics of the language. Of course, the tests can only give some degree of confidence that the prototyping tool is implemented correctly. Because of the formality of the specification language, it is possible in theory to carry out formal proofs showing that the animation tool complies with the semantics of the language. Unfortunately, since such formal definitions can be very complex and large, it will in most cases not be worth the investment, even when proof assistants are available. A more pragmatic way of checking compliance consists in showing that there is a systematic translation from the formal specification language to the animation tool. Of course, this way cannot ensure compliance but it can be made plausible. Although as yet no practical way exists to ensure or check compliance, if a specification language has formal semantics then it is potentially easier to check compliance than when no formal definition is given.

2.5 Description of the Work

This section describes and situates the work developed in this thesis in the context of the topics presented previously. As mentioned in the previous chapter, the aim of this thesis is the study and development of software tools that assist developers and users in the modelling and validation of conceptual models specified in TROLL. TROLL is a specification language with the following characteristics:

- It is a formal language, i. e. its syntax and semantics are formally defined. The syntax is given in a EBNF form. Semantics are given to TROLL specifications using different techniques: the static structure of the system is semantically described with algebraic methods, statements over states are expressed with a logic calculus, and the dynamic

structure of the system, i. e. the systems evolution, is reflected via a temporal logic which is interpreted in terms of event structures.

- It is object-oriented. TROLL models a system as a community of independent objects that communicate with each other by a synchronous action calling mechanism. Objects are encapsulated units of structure and behaviour and are described in terms of object classes. TROLL supports structuring concepts that are typical in object-orientation such as inheritance and composition.

- TROLL may easily be combined with graphical semiformal techniques such as UML. A graphical notation, called OMTROLL and based on OMT, has been specifically developed for TROLL. OMTROLL consists of different diagrams that allows developers to give a first overview of the object system. A frame of a TROLL specification can be derived from the OMTROLL diagrams. This has to be refined to a complete system specification.

Tools developed for TROLL in the context of this thesis can be classified in two groups depending on the activities they support:

- *Modelling Support:* They consist of graphical and text editors, a cross-reference generator and a syntax and static semantics checker. An automatic translation between both graphical and text notations is provided by the editors. Specifications that are syntactically correct according to the EBNF rules do not necessarily obey the typing and scoping rules given by the static semantics of the language. We have analysed these rules and implemented a static semantics checker of TROLL specifications.

- *Validation Support:* For the validation of TROLL specifications against user requirements, an animator has been constructed. This tool allows developers and users to observe, experiment and test interactively the dynamic properties of the specification through its execution in different scenarios.

The modelling and validation of a specification are an iterative process. Validation can start as soon as a first version of the specification has been modelled. The only requirement is that the syntax and static semantics of

the specification have been correctly analysed by the checker. Once the specification has been validated, it is modified in the editors accordingly. The specification can also be validated in several parts, obtaining for each part an executable prototype. The animator features are summarised as follows:

- The animator has a graphical user-friendly interface in order to encourage the participation of end users in the validation process.
- Users can explore the current state of the objects in the system and navigate through their components and specialisation aspects.
- Users can simulate the occurrence of events in the system by selecting actions in the objects to be executed.
- The execution trace, e. g. changes on attributes, interactions, etc., is shown on a console window.
- If the state transition cannot be carried out, e. g. an action precondition is not fulfilled or an integrity constraint would be violated in the next state, explanatory messages are reported to the users.
- Although the animator is used mainly for requirements validation, it also detects errors in the specification that cannot be statically checked, such as assignments of different values to the same attribute during a state transition.
- Objects are persistent, i. e. they have a lifespan that is not limited to single executions of the animator. This allows users to interrupt anytime an animation session and with the same objects' states to continue it later.
- The animator supports the validation of large complex specifications by allowing data persistence and the use of complex data types such as records and lists, and in-the-large structuring mechanisms such as inheritance and composition.

Objects created during the animation are stored in a relational database. We analyse how TROLL objects can be mapped in relational tables and implement a database schema generator for TROLL specifications.

To execute the specification, we analyse the operational meaning of state transitions in TROLL and determine an execution model using a sequential

programming language. As we will discuss in the next chapters, TROLL objects communicate with each other through a synchronous transitive *action calling* mechanism which entails the synchronisation between the life cyles of the participating objects. Aspects to take into account when executing the action chain established by the *calling* relation are parallel execution, conflicts in the assignment to attributes and variables, consistency, termination and atomicity.

Since the operational properties of TROLL partly coincide with the capabilities of object-oriented programming languages, we transform the specification into C++ code to obtain an executable prototype. We develop mapping rules from TROLL into C++ and implement a TROLL-C++ code generator. The code is generated to support the early validation of system requirements and is of a *throw-away* quality. Nevertheless, some techniques used for the automatic translation of the specification into executable prototypes can be adopted for the design and implementation of the final application.

Tools are realised as stand-alone applications and integrated in a workbench environment. The syntax graph generated by the checker is stored in a file which is directly used by other tools through a common interface. So the tools do not need to parse the specification again whenever they are called. A common graphical front-end is provided by a specification projects manager tool from which the tools can be invoked and used together. So it is easy to modify the tools contained in the workbench or incorporate new ones.

The goal of this work is the study and prototypical implementation of a software development environment to support the modelling and validation through animation of TROLL models. Typical aspects of commercial CASE tools like multiuser support, versioning of specification documents, distributed execution and control integration by inter-tool communication are outside the scope of this thesis.

2.6 Related Work

Tool support has been an important issue from the very beginning of the development of TROLL. For the first version of TROLL [JSHS91, JSHS96], a syntax and static semantics checker was implemented in [Ste92, Stö93]. First ideas about the early validation of TROLL specifications through animation were sketched in [HJSE92, HJS93]. There, the authors propose the

transformation of TROLL specifications into a kernel language that can be executed in a suitable distributed runtime environment. Previous TROLL versions to the current one included language constructs not directly or very difficult to implement such as the use of temporal logic formulae in the definition of action preconditions and integrity constraints. So they propose the translation of the specifications into an operational subset of the language. The idea was to translate the TROLL kernel language into Sather [Omo91], a language based on Eiffel [Mey92], and build a runtime system handling the distributed management of state information on different database systems, the automatic routing of event callings, and the necessary transaction protocol. Interfaces to the UNIX file system and to the relational databases Ingres and Sybase, a graphical user interface and a minimal runtime system for test purposes were implemented in [Kus93]. Nevertheless, the development did not go on and no code generation was investigated. Parallel to the development of TROLL, a reduced dialect called TROLL *light* [CGH92] was also developed. An open software development environment for the validation and verification of TROLL *light* specifications was presented in [VHG+93, GCD+95a, GCD+95b]. To verify properties of the specification, a verification calculus [Con94] was developed and implemented using a generic theorem prover. For validation purposes, an animator with basic functionalities was developed [HG94, Her95]. For the animation, specifications were first introduced into a template dictionary, which was implemented using the OODBMS ObjectStore [LLOW91]. The execution of events was carried out by an interpreter consisting of an execution module which computed the successor states [Bri93] and a term evaluator which evaluated terms and formulas [Ale93]. The object states were also stored in the database. Besides the animator did not show the execution trace, so it was very difficult to observe the behaviour of the specification, the language concepts supported by TROLL *light* were restricted with respect to the version of TROLL dealt with in this thesis. In TROLL *light*, inheritance was not supported, action preconditions could not be expressed independently of a concrete process specification, only one action in an object could happen in a state transition, and the nonexistence of an explicit specification of input and output parameters in the object actions probably indicated that information flow via action parameters was only possible in one direction[4]. More recently, the

[4]See Chap. 8 of [Har97a] for a detailed description of differences between TROLL v. 3.0 and TROLL *Light* as well as previous TROLL versions.

development of a web-based animator for TROLL *light* with similar functionalities to the previous one has been reported in [RG97a, RG97b]. For the second version of TROLL [HSJ+94, HKSH94], work towards the development of a software development environment, called *TBench*, was reported in [KHHS95a, KHHS95b]. The *TBench* project aimed at developing a distributed integrated software development environment based on the ECMA reference model [BEM92] for the specification and animation of TROLL models. Tools developed within this project were a graphical editor [Alm94], a syntax-directed editor [Emb94, Mer94] and a tool for the integration between the tools and shared access to the repository [Kle94, BP94]. Although aspects concerning the executability of TROLL v. 2.0 were investigated in [HS93, Har95] among others, no animator was developed. Based on the work reported above, we can state that the software development environment developed in this thesis represents the most complete environment, supporting graphical and textual modelling, syntax and static semantics checking and full generation of code for animation, developed for TROLL and, hence, it represents a key improvement to support and extend the use of the language.

Regarding other approaches with animation support, OASIS [LRSP98] is the most closely related to ours. Like TROLL, OASIS is a formal object-oriented language for the specification of information systems. In the OASIS approach, specifications are also modelled using graphical notations, like OO-METHOD [PIP+97] or UML [BRJ98], which are then translated to textual OASIS specifications. Semantics for OASIS specifications are given by a set of logic formulae expressed in an extension of dynamic logic which is interpreted over Kripke structures. In the last years, special emphasis has been put on the animation of OASIS specifications in concurrent environments [LSR99]. An animation environment based on concurrent logic programming is presented in [LSR97, Let99] among others. In this environment, specifications are first modelled in a graphical editor and stored in a repository. Specifications are then translated into KL1 concurrent logic programs which are compiled in order to obtain an executable prototype. A graphical user interface allows users to simulate events, that can also be read from a file, and observe the actions occurred in the objects and the reached states. Although this animation environment and the one developed in this thesis share the same aims, there are important differences between the followed approaches. Besides the execution model followed in the OASIS animator is conceived for concurrent environments, in which each object has its own execution thread, and ours for sequential ones, a main difference between both approaches lies on the com-

munication mechanism. In the OASIS animator, objects communicate with each other asynchronously while in the TROLL animator, they communicate synchronously, i. e. communication entails a synchronisation between the life cycles of the participating objects. Unlike the TROLL animator, the OASIS animator does not support inheritance, aggregation or integrity constraints, although it is planned to support them and also synchronous communication in future versions. Another animation environment for OASIS specifications based on Petri nets has been developed and reported in [SLR97, Sán00] among others. In this environment, OASIS specifications are first translated into logic dynamic formulae which in turn are translated into object oriented Petri nets (OOPNs). The generated OOPNs can be directly animated using a Petri nets animator tool. The execution traces obtained by the animation are then transformed into a legible fashion expressed in form of sequence diagrams. In this approach, the validation process takes place after the animation by analysing and contrasting the obtained sequence diagrams against scenarios developed during the requirements elicitation phase. In contrast, our validation approach is based on an interactive animation, in which users and developers simulate events and observe the system dynamics and object states during the animation session. Parallel to the development of these animators, another research direction within OASIS has been the development of a CASE environment supporting OO-METHOD, a methodology based on OASIS [PIP+97, PPIG98, PCR99]. The OO-METHOD CASE allows developers to specify graphically the system and, using OASIS as intermediate language, generates automatically code in several programming languages. Unlike in the TROLL animator, the code generation in the OO-METHOD CASE is not oriented to validation purposes but rather to obtain the final application.

Another language with animation support is ALBERT II [DB95, DB97]. ALBERT II is a formal requirements specification language designed to specify the requirements of distributed real-time systems. An ALBERT II specification consists of agents (active objects) which can perform or suffer actions that may change their states (denoting either physical or informational characteristics), and whose admissible behaviours are restricted by means of constraints. The semantics of an ALBERT II specification are given in a real-time temporal logic called ALBERT_{KERNEL}. Work towards the animation of ALBERT II specifications has been reported in [Hey97, HD98] and elsewhere. Specifications used in the animator are edited in a graphical environment with syntax and type checking facilities. The animator allows step-by-step testing

of scenarios against the specification using an interpretation algorithm. Unlike the TROLL animator, the main challenges of the ALBERT II animator are to provide a distributed tool which allows stakeholders to cooperatively animate a specification and to deal with undeterminism.

TROLL has its roots in the specification language OBLOG [SSE87]. While TROLL aims have been focussed on the academic realm, special emphasis has been put on the development of OBLOG as a commercial product. The current version of the language combines a graphical notation, which is UML compliant, with a textual one. The textual notation is mainly used for the design details. A commercial CASE environment has been developed supporting the edition, validation, reverse engineering and automatic generation of application code and documentation of OBLOG specifications[5]. A main feature of the OBLOG CASE environment is the use of customisable transformation rules in the generation of code allowing the support for multiple target languages and architectures. For validation purposes and making use of the Java generation rules, a Java prototype can be automatically generated from the specification. An animator allows users to interact with the prototype visualising the results of the animation in a form of interaction diagrams. Although OBLOG was originally developed with a sound theoretical basis, it is unclear, regarding the information obtained from its web page, whether the commercial version still maintains the theoretical backgrounds.

Nowadays there exists a great variety of formal specification languages supported by software tools[6]. The spectrum of these tools ranges from editors, syntax and type checkers to verification and validation tools. Regarding validation support, the model-based languages Z [Spi92] and VDM [Jon90] are the main languages for which animation tools have been developed (see, for instance, [MJHB98, O'N92, ELL94]). A general problem of current animation environments is that due to the abstract notations of the specification languages and the complex way their execution behaviour is usually presented to users, they are more oriented to developer validation than user validation [Özc98].

A large number of commercial CASE tools supporting object oriented analysis and design techniques is currently available[7]. Although these tools

[5]The first commercial version of the OBLOG toolset was released in 1999. The OBLOG web site is located at http://www.oblog.com.

[6]For an overview of tool support for formal methods, we refer the reader to the web page http://www.comlab.ox.ac.uk/archive/formal-methods.html

[7]For a good overview of current OOAD CASE tools we suggest the reader to consult

include powerful graphical editors, most of them supporting UML notations, and automatic code generation, which ranges from generation of headers and program skeletons to complete application code, only a few of them provide specific support to requirements validation. BridgePoint[8] provides a "Model Verifier" module which allows Shlaer-Mellor models to be executed for debugging prior to code generation. Some CASE tools such as ObjecTime[9], Rhapsody[10] and BetterState[11] that model behaviour with state machines allow the animation of these models by adding code, e. g. in C++ or in a proprietary language, to represent actions that occur on the state transitions. In Rhapsody and ObjecTime, models are validated by contrasting the execution traces against scenarios expressed as sequence diagrams (in ObjecTime, this can be done automatically).

2.7 Summary

A main activity in the conceptual modelling process is the validation of the specification against the informal user requirements. Validation requires the participation of users because they provided the original requirements. A limitation of formal specifications is that they cannot be readily understood by users unless they have been specially trained. A technique for making easier the user participation in the validation process consists in exploiting the executability of the specifications. By animating the specification users can observe the dynamic behaviour of the specification in different scenarios to see if it meets their real needs. This thesis concerns with the study and development of software tools that assist developers and users in the modelling and validation through animation of conceptual models specified with TROLL. The tools consist of graphical and text editors, a cross-reference generator, a syntax and static semantics checker, and an animator. Objects created during the animation are stored in a relational database. To this end, we analyse how the static structure of TROLL objects can be mapped into relational tables. In order to execute the specification, we analyse the operational meaning of state transitions in TROLL, give an execution model

the web page http://www.cetus-links.de/oo_ooa_ood_tools.html

[8] http://www.projtech.com

[9] http://www.objectime.com

[10] http://www.ilogix.com

[11] http://www.isi.com

and develop mapping rules from TROLL into C++. The executable prototypes are generated automatically. In the animator, users can observe the current state of the objects, navigate through their relationships, simulate the occurrence of events and observe the execution traces. With the work developed in this thesis, we show that far from being a handicap, formal specifications represent a key improvement in the validation process by allowing the construction of powerful validation tools such as specification animators.

Chapter 3

The TROLL Approach to Conceptual Modelling

A formalism or methodology for specifying information systems may be described as the symbiosis of three concepts: a language, a method for using the language and a set of tools to support developers in the modelling process. In this chapter, we introduce the textual object-oriented specification language TROLL together with its graphical part OMTROLL and illustrate the language concepts by example. After a brief introduction in Sect. 3.1., we present the modelling with OMTROLL and TROLL in Sect. 3.2 and Sect. 3.3 respectively. The formal semantics underlying the TROLL language are briefly described in Sect. 3.4.

3.1 Introduction

TROLL (Textual Representation of an Object Logic Language) is a formal object-oriented specification language for specifying information systems at a high level of abstraction. TROLL belongs to the OBLOG (OBject LOGic) family of languages. OBLOG was first presented in the publication [SSE87] as a language for specifying software systems as a society of interacting objects. The semantic foundations of OBLOG are based on the concept of abstract object types, an extension of abstract data types to cover objects and object systems. Since 1987, OBLOG and its supporting CASE environment have been further developed [CSS89, Esp93] becoming a commercial product. At the university and based on OBLOG there are two

related developments: TROLL and GNOME [SR94]. TROLL is currently in its third version [DH97, Har97a, GKK+98]. Previous versions are TROLL v. 0.01 [JSHS91, JSHS96], TROLL *light* [GCD+95b] and TROLL v. 2.0 [HSJ+94, HKSH94]. In contrast to older versions, TROLL version 3.0 is devoted to modelling distribution issues and has also been designed with the aim of executability. The design of TROLL v. 3.0 has been significantly influenced by the experiences gained in an industrial project in which TROLL has been used as the modelling language. The project started in 1994 in cooperation with PTB (Physikalisch-Technische Bundesanstalt[1]). It aims at developing an information system which supports the testing and certifying of explosion proof electrical equipment. The system is called CATC (Computer Aided Testing and Certifying) and includes different subsystems based on the certifying process [HDK+97, KG98].

TROLL has also a graphical part called OMTROLL. The graphical notation serves for giving a first overview of the system and as a means of communication between users and developers. The textual notation incorporates all language concepts, whereas the graphical notation covers only part of the concepts. From OMTROLL a frame of a textual TROLL specification can be derived. The textual specification has to be refined to a complete system specification. Fig. 3.1 depicts the TROLL design process. A method

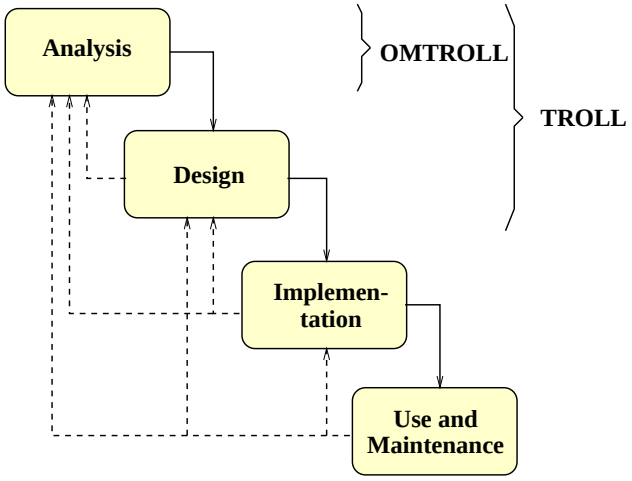


Figure 3.1: The (OM)TROLL Design Process

[1]German Federal Institute of Weights and Measures.

for using the language concepts is given in [Kow96, DH97]. The method consists of a series of steps and rules that guide developers during the modelling process. In the next sections, we present OMTROLL and TROLL by modelling a simplified extract from the CATC system and according to these steps. Next, we describe briefly the CATC system.

Example 3.1.1 *(CATC System)* The CATC project aims at designing an information system for supporting the various activities of group 3.4 within PTB. The group is concerned with testing and certifying electrical equipment, e. g. motors, switches, heaters, which are to be used in explosion-hazardous areas. The certifying process for electrical equipments is composed of the following parts:

- *Administration management* includes the preliminary screening of documents concerned with the company producing or providing the electrical equipment. Such documents contain, for instance, the name and address of the company and the technical description of the equipment.

- *Design approval* includes the assessment of design papers for the equipment based on descriptions and its accordance with the European standards. Here the staff creates a check list including the necessary experiments that have to be done by the operators.

- *Experimental tests* are performed by the operators in the test lab and store all relevant data. There are different tests depending on the kind of electrical equipment (for example, temperature tests for heaters and pressure test for motors). The tests are done on samples sent by the company. Once the tests have been performed, the staff reads the results and decides whether more tests are necessary. Otherwise, the certification may be issued.

The example used for illustrating the TROLL concepts along this thesis concentrates on the experimental part of the system. The testing process is a cooperation between staff and operators. Both of them have to access to the test results. The operators produce and store the test results while the staff reads and interprets them. Due to European standards, every application should be tested by more than one experiment to avoid any mistake. While the experiments for one application are running, the staff may set up a next experiment. □

3.2 OMTROLL

The use of graphical notations in the specification of software systems such as entity-relationship (ER) models and data flow diagrams (DFD's) has always found a widespread acceptance. Especially in the last years, a great variety of graphical object-oriented analysis and design techniques such as *UML* [BRJ98] have aroused and become very popular. The reason for this success is that graphical notations are usually more intuitive and easier to understand than textual notations. The TROLL modelling process combines the simple expressiveness of these popular notations with the formal nature of TROLL taking the advantages from both of them. OMTROLL [WJH+93, JWH+94, DH97] consists of different diagrams for modelling a system. The idea is to use OMTROLL at the very beginning of the modelling process in order to obtain an abstract outline of the system. The information which is represented in the OMTROLL diagrams becomes the underlying information used for the TROLL specification. OMTROLL is based on the popular *Object Modelling Technique (OMT)* method developed by Jim Rumbaugh [RBP+91][2].

The OMTROLL graphical specification allows the initial design to be realised through the use of several *system views*: community model, data type model, object model, dynamic model and communication model. Each view is associated with a specific diagram:

- The *community diagram* gives an overview of the object classes and their respective relationships to each other.
- The *data type diagram* represents user defined data types which are defined over standard data types.
- The *object class declaration diagram* allows the developer to describe each object class in further detail. That is, the developer can define attributes and operations relevant to each class.
- The *behaviour diagram* defines the life cycle of the objects. The main aim of the behaviour diagram is to determine preconditions of actions and dependencies between action executions.

[2]Recently *OMT* has been combined with other popular object-oriented design techniques becoming *UML (Unified Modelling Language)*

- The *communication diagram* establishes the communication structure between the objects of the system. This diagram is not present in *OMT* and can be compared with *Fusion* diagrams [CAB+94].

In the following, we describe each of these diagrams by example. Appendix A.1 describes the syntax of the OmTroll diagrams. A more detailed description of OmTroll can be found in [DH97].

Community Diagram

The community diagram represents the structure of the objects in the system. An object system is defined as a community of concurrent interacting objects. Objects are classified into object classes. An object class is a generic template for describing objects with similar properties and common behaviour. Object classes may be aggregated or specialised. They may have other object classes as components (*part-of relationship*) as well as specialisations (*is-a relationship*). A specialisation (subclass) inherits and extends the properties of a base class (superclass). The next example illustrates the community diagram of the CATC system.

Example 3.2.1 *(Community Diagram of CATC)* The community diagram is depicted in Fig. 3.2. The system has two kind of objects: `IG34` and `Users`

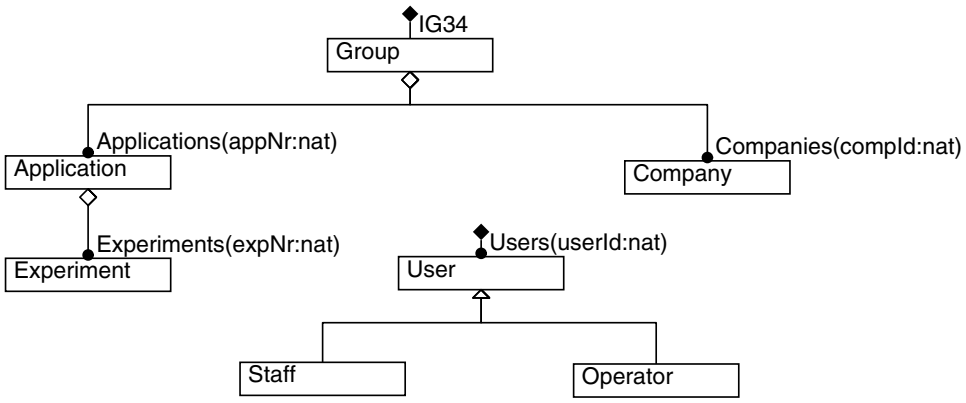


Figure 3.2: OmTroll Community Diagram

which are described by the classes `Group` and `User` respectively. `IG34` represents the information managed in the certifying process by group 3.4 of PTB.

It is composed of `Applications` and `Companies`. Components are represented by a diamond and the filled dots at the top of the object classes denote multiple components. An `Application` may have several `Experiments` as components. `Users` may be specialised in `Staff` and `Operator`. Specialisation is denoted by a triangle. □

Data Type Diagram

User-defined data types are defined in the data type diagram. Apart from predefined data types, TROLL allows the use of constructors such as *list* and *record* for building new data types. Data type definitions will be presented in more detail in the next section.

Example 3.2.2 *(Data Type Diagram of CATC)* Fig. 3.3 shows some data type definitions of the CATC system. Experiments may be performed in different labours. This is depicted by the enumeration type `labours`. The type of users is represented by `users_type`. The setup for an experiment (`msset`) is defined as a record consisting of a pressure value which will be put in the device during a predefined time. The experiment results (`msresults`) are defined by a list of real numbers that represent the final pressures in the device. Finally, a company address is represented by a record containing the street, number and city where the company is located. □

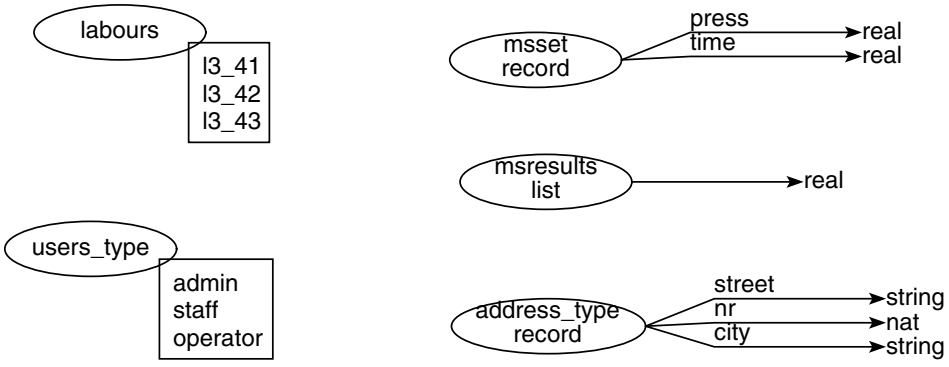


Figure 3.3: OMTROLL Data Type Diagram

Object Class Declaration Diagram

Classes are further specified in object class declaration diagrams. Attributes and actions relevant to each class are declared here. The declaration of an attribute is given by its type and additional properties. Attributes may be initialised (\i), constant (\c), hidden (\h), optional (\o) or derived (\d). Actions may have input and output parameters, so objects can exchange information with each other. Apart from update actions, there are birth and death actions which create and destroy objects respectively. Birth actions are marked with a "*" and death actions with a "+". Actions may also be hidden. Object declaration diagrams and the community diagram define the structural part of the system and are usually represented together.

Example 3.2.3 *(Object Class Declaration Diagrams of CATC)* The object class declaration diagrams of `Application` and `Experiment` are depicted in Fig. 3.4. Attributes of the class `Application` are `company, labour, app_date` and `nextExpNr`. `company` is an object valued attribute that makes

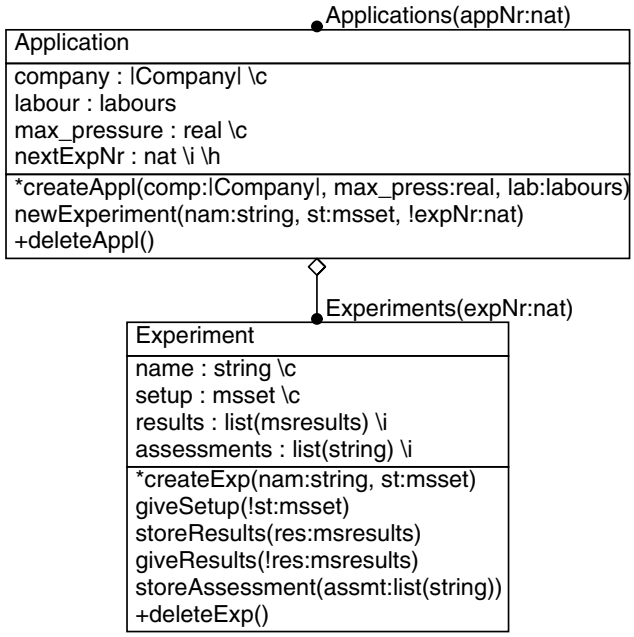


Figure 3.4: OmTroll Object Class Declaration Diagram

reference to an object of class `Company`. `company` is a constant attribute, i. e. its value is set at the beginning and never changes. `labour` denotes the labour to which the application is assigned. `max_pressure` indicates the maximum pressure which can be set up in the experiments for the application. `nextExpNr` represents the identity number of the next experiment to be performed to the application. It is initialised and hidden, i. e. it is only visible inside the class `Application`. The birth action of an `Application` is `createAppl` and its input parameters represent values for the class attributes which are not initialised. The action `newExperiment` represents the creation of a new experiment for the application. It has as input parameters the name and setup for the experiment to be created and as output parameter the identity number of the experiment. Output parameters are denoted by prefixing a "!" before its name. The class `Experiment` has as attributes the name, the setup, the results and the list of assessments of the experiment. The actions of `Experiment` are explained below in the description of the behaviour diagram. □

Object Behaviour Diagram

Object behaviour diagrams are state transition diagrams representing the local behaviour over time of the objects. A state diagram is a directed graph whose nodes represent states and whose arcs represent transitions between states. Arcs are labelled by the actions which cause the transition and may also contain enabling preconditions. TROLL does not explicitly support the definition of states. So states in behaviour diagrams are used for auxiliary purposes. As a consequence of it and unlike the rest of OMTROLL diagrams, object behaviour diagrams cannot be directly translated into TROLL. This issue will be discussed later in the presentation of the OMTROLL editor in Sect. 7.2.2.

Example 3.2.4 *(Object Behaviour Diagrams of CATC)* Fig. 3.5 shows the object behaviour diagrams corresponding to an experiment and an operator. Analogous to birth and death actions, there are initial and final states. A dot represents an initial state, a circled dot depicts a final state. The life cycle of an experiment starts with its creation (`createExp`). Setup information is then requested by the operator in charge in order to set up and start the experiment (`giveSetup`). Next, the experiment results are stored (`storeResults`) and returned by request (`giveResults`). Once the results

are given, assessments about the results may be stored (`storeAssessment`). Finally, an experiment is deleted through its death action (`delExp`) going to its final state. Operators are created in the system by the execution of the birth action (`login`). They ask then the setup information for an experiment (`askSetup`), perform the experiment (`startExperiment`) and store the results (`giveResults`). Once the results are stored, operators may either request the setup information for a new experiment (`askSetup`) or exit the system (`logout`). □

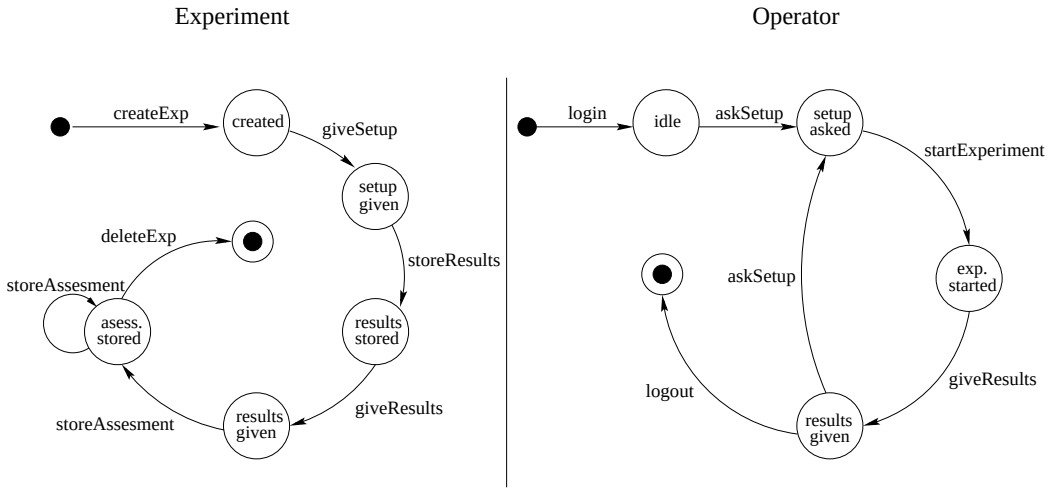


Figure 3.5: OmTroll Object Behaviour Diagram

Communication Diagram

Interactions among objects are depicted in communication diagrams. Objects interact with each other through action calling. Communication diagrams show for each action its calling relations. Action calling may be either local, i. e. inside a complex object, or global, i. e. among two concurrent objects. The latter means a synchronisation relation in the life cycle of the objects. In Troll, action calling is directed and synchronous.

Example 3.2.5 *(Communication Diagram of CATC)* The communication diagram depicted in Fig. 3.6 shows the communication among objects participating in the testing process. The creation of an experiment entails the

synchronous execution of three actions: `createExperiment` in `Staff` which calls the action `newExperiment` in the `Application` with identity `AppNr` which in turn calls the birth action of the component `Experiment` with identity `expNr`. The `Operator` has to know under which pressure and time the test has to be done. To this end, (s)he asks for the setup values (`askSetup`) and gets the answer from the action `giveSetup` in `Experiment`. After performing the test, (s)he stores the results in `Experiment`. On the other hand, the `Staff` asks for the results, checks them and store his/her assessments.

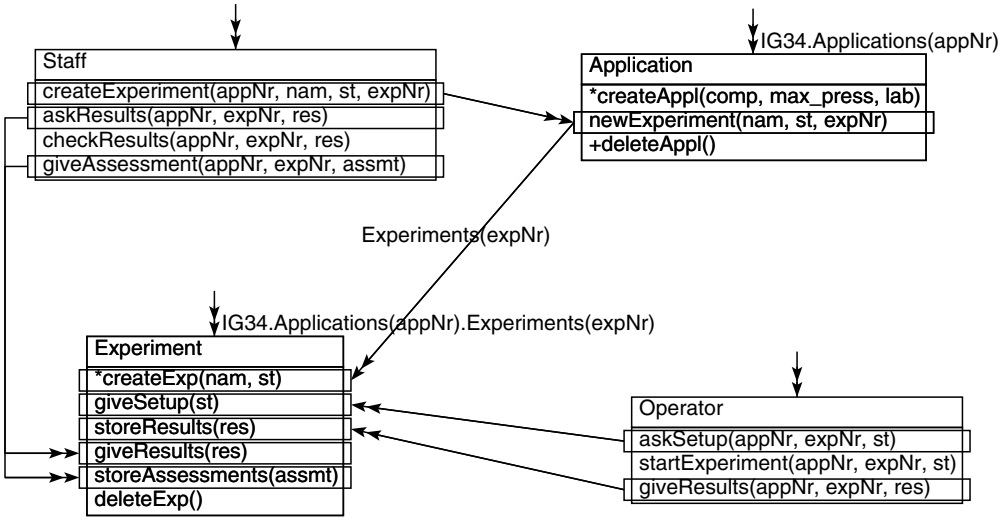


Figure 3.6: OMTROLL Communication Diagram

□

3.3 TROLL

As mentioned in Sect. 3.1, a frame of a TROLL specification can be derived from the OMTROLL diagrams. The textual specification has to be refined to a complete system specification. Fig. 3.7 shows the structure of a TROLL specification. It consists of four parts: data type definitions, object class specifications, object declarations and a global behaviour specification.

user defined data types

data type <*type_name*>=<*type*>

class declarations

object class <*class_name*>
  aspect of <*superclass*> **if** <*action*>...
  components
    <*comp*>(<*type*>) :<*class*> [**hidden**]
  attributes
    <*attr*>:<*type*> [**constant, optional, initialized, derived, hidden**]
  actions
    <*act*>(p_1:t_1, ..., p_n:t_n) [**hidden**]
  behavior
    <*act*>(p_1, ..., p_n)
      onlyif(<*formula*>)
      var <*var*>:<*type*>, ...
      do
        assignments, callings, conditionals, iterations
      od
  constraints
    [**initially**] <*formula*>,...
end

object declarations

objects <*obj*>(<*type*>) :<*class*>

global interaction definitions

behavior
    <*act*>(p_1, ..., p_n)
      do
        ...
      od

Figure 3.7: Structure of a TROLL specification

- *Data Type Definitions:* This part includes the definition of user data types.
- *Object Class Specifications:* They define prototypical object descriptions. An object class consists of a signature and a behaviour definition. Attributes and actions are declared in the signature part. The behaviour of an object is defined by the description of rules for its actions and the declaration of integrity constraints. Preconditions may also be declared to restrict the occurrence of actions. Possible action rules are assignments, action callings, conditionals and iterations. Object classes may be complex. They may have other object classes as components (*part-of relationship*) as well as specialisations (*is-a relationship*).
- *Object Declarations:* Objects are declared over object classes. These declarations describe the set of concurrent objects in the system.
- *Global Behaviour Specification:* This part describes the interactions between concurrent objects in the system. Objects may communicate with each other by action calling where data flow is modelled via input/output parameters. All actions involved in an interaction are understood to take place simultaneously.

The TROLL syntax can be found in Appendix A.2. For a more complete description of TROLL see [DH97, Har97a]. In what follows, we describe each of these parts in further detail. To illustrate the concepts, we use the example of the CATC system. The complete TROLL specification of the example can be found in Appendix B.

Data Type Definitions

TROLL has a range of predefined data types:

nat, int, real, money, bool, string, char and *date*.

In addition, TROLL allows the use of data type constructors. Users can apply these constructors to predefined or user-defined data types in order to construct complex data types. The constructors are:

$$list(t),\ set(t),\ bag(t),\ record(id_1{:}t_1,\ldots,id_n{:}t_n),\ map(id{:}t_1,t_2),\ enum(id_1,\ldots,id_n)$$

Predefined generic operations for the type constructors such as *concatenation, head, tail* etc. of lists are also available. We illustrate the definition of data types by example.

Example 3.3.1 *(User-Defined Data Types)* Data type definitions can directly be derived from OMTROLL data type diagrams. In Fig. 3.3 we illustrated the OMTROLL data type diagram representing the user-defined data types of the CATC system. The corresponding data type definitions in TROLL are:

```
labours      = enum(l3_41,l3_42,l3_43);
users_type   = enum(staff,operator);
msset        = record(press:real,time:real);
msresults    = list(real);
address_type = record(street:string,nr:nat,city:string);
```

□

Besides predefined and complex data types, there are object-valued data types. Each object class specification defines implicitly an identification data type. Identification data types represent references to objects. They are denoted by the class names enclosed by vertical lines (|<class_name>|). For instance, in example 3.2.3 we defined in the class `Application` the attribute `company` as a reference to an object of class `Company` (|Company|).

Object Class Specifications

A class specification describes the local structure and behaviour of objects whose properties are similar. Classes are generic object templates. They are not containers of object instances. As we will see later, objects are declared and identified outside the classes. An object class specification is composed of two parts: a *definition* and an *implementation* part. The definition part consists of three parts (all optional): a signature declaration for attributes and actions, a component declaration and a specialisation declaration. On the other hand, the implementation part includes operation definitions for actions and constraints. The former describes what are the effects of the actions and under which preconditions they may occur. The latter can be used to formulate static integrity constraints for objects. Next, we describe briefly each of these parts.

Signature Declaration

An attribute declaration is given by its name, data type and special features. An attribute may be:

- *constant:* The value of a constant attribute is set at the birth of the object and remains fixed throughout the object's lifetime.
- *optional:* The value of an optional attribute does not have to be fixed at the object's creation and can be fixed later.
- *initialized:* Initialised attributes have a fixed value at its object's birth. This value is independent of any object generation.
- *derived:* The value of derived attributes are determined by values of other attributes.
- *hidden:* Hidden attributes are only visible in the classes in which they are declared.

An action declaration is given by its name and parameters. In addition, an action may be declared hidden. A hidden action is only known in the class in which it is declared. Birth and death actions create and destroy objects respectively. The signature declaration can be derived from the object class declaration diagram. Additionally, initial values for initialised attributes and derivation rules for derived attributes must be specified. We illustrate the declaration of attributes and actions in example 3.3.2.

Component Declaration

In TROLL, components are not shared by objects, i. e. a component exclusively belongs to an object. Components do not have an autonomous existence. They depend existentially on the composite object to which they belong. Components may be single or multiple. The latter are identified by a parameter. The signature of a composite class is extended by the signatures of its components. This means that visible attributes and actions of component classes are also visible in the composite class. Attributes may only be assigned in the class in which they are declared. So composite classes have only read-access to the attributes of their components. In order to restrict

the visibility of a component to the class in which it is declared, the component may be declared as *hidden*. The next example shows the signature and component declarations of a class.

Example 3.3.2 *(Signature and Component Declarations)* Fig. 3.2.3 depicted the object class declaration diagram corresponding to an `Application` in the CATC system. The corresponding TROLL specification is defined as follows:

```
object class Application
  components
    Experiments(expNr:nat):Experiment;
  attributes
    company : |Company| constant;
    labour : labours;
    max_pressure : real constant;
    nextExpNr : nat initialized 1, hidden;
  actions
    * createAppl(comp:|Company|, max_press:real, lab:labours);
    newExperiment(nam:string,st:msset,! expNr:nat);
    + deleteAppl;
  ...
end;
```

□

Specialisation Declaration

A class may be the specialisation of another class. A specialisation represents a particular *aspect* of a basis class. It adds new properties to the basis class. In TROLL, specialisations are static. That is, objects are specialised at the moment they are created and cannot get new aspects or give up existing ones during their lifetime. Multiple inheritance is not allowed, i. e. a class may only be the specialisation of one basis class. However, specialisations of a basis class do not have to be disjoint, and hence an object may have several specialisation aspects. A specialisation declaration is given by the superclass name and a set of birth actions belonging to the superclass. If necessary, the set of birth actions may have formulas in order to restrict the specialisation conditions. The signature of the basis class is embedded in the specialisation classes. Thus attributes and actions of the basis class are visible in the specialisations. Furthermore, the behaviour of an action of

the basis class may be extended in specialisations. Next, we illustrate the declaration of specialisation classes by example.

Example 3.3.3 *(Specialisation Declaration)* As mentioned in example 3.2.1, users of the CATC system may be specialised in staff and operators. This is depicted as follows:

```
object class User
 ...
 actions
  * login(name:string,dat:date,user_t:user_type);
 ...
end;
object class Staff
 aspect of User if login(name,dat,user_t) and user_t=staff
 ...
end;
object class Operator
 aspect of User if login(name,dat,user_t) and user_t=operator
 ...
end;
```

An object of class `User` will also be of class `Staff` if it is created by the birth action `login` and the input parameter `user_t` is set to `staff`. On the other hand, the object will be of class `Operator` if it is also created by the action `login`, but `user_t` is set to `operator`. □

Behaviour Specification

The behaviour specification of an object class contains the definition of operations to be executed by the occurrence of actions and the definition of integrity constraints. Unlike other parts of a TROLL specification, the behaviour part cannot completely be obtained from the OMTROLL diagrams. Although local communication can be derived from the communication diagram, the rest of the behaviour specification has to be directly written in TROLL. Nevertheless, object behaviour diagrams may help in this step. Preconditions can be defined in order to restrict the occurrence of actions. An action can only occur if its precondition is satisfied. In TROLL, a state transition can only happen if all actions involved in the transition can occur, i. e.

all their preconditions are fulfilled. We will discuss TROLL state transitions later in Sect. 6.1. Local variables may also be declared for each action. The effect of an action is defined by action rules. There are four kind of action rules:

- *valuation rules:* Values can be assigned to attributes, output parameters and local variables.
- *calling rules:* Actions may call other local actions. Local actions also include actions defined in component and basis classes. Communication between components can be specified in the composite. When an action is called, data terms with defined values have to be specified for each input parameter. For the output parameters, local variables may be defined.
- *iteration rules:* Multicalls can be specified by iteration rules. A multicall is the simultaneous call of several actions usually belonging to component objects.
- *conditional rules:* Different behaviour can be specified depending on a condition. The fulfilment of the condition may be determined by the current values of attributes and input parameters. Thus its satisfaction may be state-dependent.

Since the signatures of components are embedded in the composite, the behaviour of component actions can also be defined in the complex class. Similarly, the behaviour of actions belonging to a superclass can be defined in its specialisations. In both cases, the behaviour rules of an action are never overridden but extended. We will come back to this later.

Apart from defining operations for the actions, the behaviour of an object can be given by the definition of constraints. Integrity constraints restrict the possible values of attributes. They can be defined as *initially* meaning that the constraint has to be satisfied at least in the first state of the object's life. In other case, constraints are defined as *static*, i. e. they have to be satisfied in every state. Constraints are defined using first order logic formulas that may be quantified over finite sets. The following example depicts the behaviour specification of a class.

Example 3.3.4 *(Behaviour Specification)* The signature declaration of the class `Application` was presented in example 3.3.2. There, two actions were

declared: the birth action `createAppl` and the action `newExperiment`. The local behaviour definition of these actions and the definition of constraints for the class attributes are defined as follows:

```
object class Application
  ...
  behavior
    createAppl(comp,max_press,lab)
    do
      company := comp,
      max_pressure := max_press,
      labour := lab
    od;
    newExperiment(nam, st, expNr)
    onlyIf (st.press <= max_pressure)
    do
      Experiments(nextExpNr).createExp(nam,st),
      expNr := nextExpNr,
      nextExpNr := nextExpNr+1
    od;
  constraints
    nextExpNr<=11;
  ...
end;
```

The action `createAppl` creates an application and assigns the values of its input parameters `comp`, `max_press` and `lab` to the attributes `company`, `max_pressure` and `labour` respectively. The attribute `nextExpNr` is not assigned in the birth action because its initialisation value is defined in the attribute declaration. The action `newExperiment` creates a new experiment for an application. The occurrence of `newExperiment` is limited by a precondition that indicates that the initial pressure given in the experiment's setup (`st.press`) must be less than or equal to the allowed maximum pressure (`max_pressure`). `newExperiment` calls the birth action (`createExp`) of the component `Experiments` whose identifier number is determined by the attribute `nextExpNr`. The identifier number of the new experiment is returned in the output parameter `expNr`. The next experiment number is increased by 1. Finally, an integrity constraint denotes that `nextExpNr` has to be less than

or equal to 11 in every state, i. e. there cannot be more than 10 experiments for each application. □

Object Declarations

Once data types and object classes have been declared, the next step is to declare the instances of the object system. An object declaration consists of the object's name, a parameter in case of multiple objects, and the name of the class that describes the structure and behaviour of the objects. Object declarations can be directly derived from the OMTROLL community diagram.

Example 3.3.5 *(Object Declarations)* The object declarations corresponding to the CATC community diagram depicted in Fig. 3.2 are:

```
objects IG34:Group;
objects Users(userId:nat):User;
```
□

Global Behaviour Specification

Global interactions between concurrent objects are defined outside the classes. In this way, object class specifications are independent of the system in which they are used and can therefore be easily reused. Data flow among actions is specified by input/output parameters. Unlike the local behaviour specification in object classes, attributes cannot appear in the global behaviour specification. Global interactions can be derived from the OMTROLL communication diagram.

Example 3.3.6 *(Global Behaviour Specification)* Following the CATC communication diagram of Fig. 3.6, `Operator` interacts with `IG34` as follows:

```
behavior
 Operator(Users(userId)).askSetup(appNr,expNr,st)
 do
  IG34.Applications(appNr).Experiments(expNr).giveSetup(st)
 od;
 Operator(Users(userId)).giveResults(appNr,expNr,res)
 do
  IG34.Applications(appNr).Experiments(expNr).storeResults(res)
 od;
 ...
end;
```
□

3.4 Formal Semantics

In this section, we explain briefly some aspects of the formal semantics underlying the TROLL language. We focus on features for specifying inter-object communication. Further details, including in-the-large features such as composition and inheritance, can be found in [Ehr99, EH96] among others. As in the previous sections, concepts are illustrated by means of the CATC example.

A TROLL specification determines structural as well as dynamic aspects of the system. The system structure is given by the system object instances together with their attributes and actions as described in the corresponding object classes. The system dynamics are given by initialisation declarations, action preconditions, action effects on the attributes, global interactions among the objects, constraints, etc. A TROLL system specification is formalised as a pair $SysSpec = (\Sigma_I, \Phi)$ where Σ_I is a system signature, describing the structural part of the system, and Φ is a set of axioms in logical formulae, describing the dynamic aspects of the system. Axioms are expressed in a distributed temporal logic and interpreted over labelled prime event structures. The translation of all TROLL language concepts in formulae of this logic is described in [Har97a].

Let $\Sigma = (S, \Omega)$ be a signature where S is a set of data and object sorts, i. e. $S = S_D \cup S_O$, and Ω is an $S^* \times S$-indexed family of sets of operation symbols. A *system signature* is given by $\Sigma_I = (Id, At, Ac)$ where Id is a set of object identifiers (the system object instances), At is an $Id \times S$-indexed family of sets $At = \{At_{i,s}\}_{i \in Id, s \in S}$ of attribute symbols, and Ac is an $Id \times S^*$-indexed family of sets $Ac = \{Ac_{i,x}\}_{i \in Id, x \in S^*}$ of action symbols. For an arbitrary $a \in At_{i,s}$, a is an attribute of object i and type s. For an arbitrary $b \in Ac_{i,s_1 \ldots s_n}$, b is an action of object i with n parameters of sorts $s_1 \ldots s_n$. We illustrate these concepts in the next example.

Example 3.4.1 *(System Signature of CATC)* Let `1, 2, ...` be user numbers, `1, 2, ...` be application numbers and `j` be an arbitrary operator user number. The CATC system signature is given by $\Sigma_{CATC} = (Id, At, Ac)$ where for instance

Id = {`IG34, Users(1), Users(2), ...`}
$At_{\texttt{IG34,labours}}$ = {`IG34.Applications(1).labour, ...`}
$At_{\texttt{IG34},string}$ = {`IG34.Applications(1).Experiments(3).name, ...`}

$Ac_{\texttt{IG34},string\ \texttt{msset}\ nat} = \{\texttt{IG34.Applications(1).newExperiment, ...}\}$
$Ac_{\texttt{Users(j)},nat\ nat\ \texttt{msset}} = \{\texttt{Users(j).askSetup}\}$

The attributes and actions of the components of `IG34`, i. e. its applications, are imported in its signature. □

System axioms are described in a distributed temporal logic called DTL[3]. DTL is an extension of linear temporal logic based on n-agent logics [LRT92]. Each object $i \in Id$ has a local logic DTL_i. A local logic DTL_i allows the object i to make assertions about system properties from its local viewpoint. Such system properties can be made based on the communication with other objects. DTL is defined as follows:

$$\text{DTL} ::= \{\text{DTL}_i\}_{i \in Id}$$

$$\text{DTL}_i ::= i.H_i \mid i.C_i$$

$$H_i ::= \text{ATOM} \mid (H_i \Rightarrow H_i) \mid (H_i\ \mathcal{U}\ H_i) \mid (H_i\ \mathcal{S}\ H_i)$$

$$\text{ATOM} ::= false \mid T_\Sigma\ \theta\ T_\Sigma \mid AT_\Sigma\ \theta\ T_\Sigma \mid \triangleright AC_\Sigma \mid \odot AC_\Sigma$$

$$C_i ::= \mathcal{C}_i \Rightarrow j.\mathcal{C}_j \qquad \text{for some } j \in Id, i \neq j$$

The local logic of object i is split into a local *home* logic H_i and a *communication* logic C_i. H_i is a first order linear temporal logic. T_Σ, AT_Σ and AC_Σ represent data, attribute and action terms respectively. An atomic formula of H_i can be the boolean constant $false$; the predicate θ applied to two data terms or to an attribute and data term, where θ is a comparison predicate (e. g. $=, \leq, \ldots$); the predicate $\triangleright$ (enabling) applied to an action term; or the predicate $\odot$ (occurrence) applied to an action term. A formula in H_i can be obtained by applying successively the connective $\Rightarrow$ and the temporal operators $\mathcal{U}$ (until) and $\mathcal{S}$ (since) to atomic formulae. The other temporal operators like *next* X, *sometime in the future* F and *always* G can be derived from these (cf. [ECSD98] for details). The communication logic C_i allows to express communication among several objects from the local viewpoint of object i. Formulae in $\mathcal{C}_i$ contain occurrences of actions for object i like, e. g. $\odot i.a \wedge \triangleright i.b$. The next example illustrates some axioms of the CATC system. To increase their readability we will omit the local object identity in the formulae.

[3]Currently, this logic has been renamed into $\texttt{D}_0$ in contrast to $\texttt{D}_1$, a more high-level logic but no more expressive than $\texttt{D}_0$. Both logics are described in [ECSD98]. The translation from $\texttt{D}_1$ into $\texttt{D}_0$ is formally described in [EC00].

Example 3.4.2 *(System Axioms of CATC)* In the following, we present some examples of axioms derived from the CATC specification.

The next axiom describes the effect of the birth action of an application on the attributes:

```
⊙ IG34.Applications(appNr).createAppl(comp,max_press,lab) ⇒
    IG34.Applications(appNr).nextExpNr = 1 ∧
    IG34.Applications(appNr).company = comp ∧
    IG34.Applications(appNr).max_pressure = max_press ∧
    IG34.Applications(appNr).labour = lab
```

The following axiom translates the *constant* feature of the attribute `max_pressure` of an application, i. e. after a maximum pressure has been set, it will remain unchanged for the rest of the application's life:

```
IG34.Applications(appNr).max_pressure = p ⇒
    (G IG34.Applications(appNr).max_pressure = p 𝒰
    ⊙ IG34.Applications(appNr).deleteAppl)
```

The precondition of the action `newExperiment` of an application is translated as follows:

```
▷ IG34.Applications(appNr).newExperiment(nam,st,expNr) ⇒
    st.press <= IG34.Applications(appNr).max_pressure
```

The next formula denotes a synchronous communication between an operator instance and `IG34`:

```
⊙ Operator(Users(2)).askSetup(appNr,expNr,st) ⇒
    ⊙ IG34.Applications(appNr).Experiments(expNr).giveSetup(st)
```

□

DTL is interpreted over labelled prime event structures (cf. e. g. [NPW81]). Event structures represent a simple description of distributed computations as event occurrences together with a causal and a conflict relation between them. The causal relation implies a (partial) order among event occurrences, while the conflict relation allows to express choice. Events in conflict cannot belong to the same system or object run. Formally, a prime event structure is a triple $E = (Ev, \rightarrow^*, \#)$, where Ev is a set of events, $\rightarrow^*$ and $\#$ are binary relations on events called causality (a partial order) and conflict (irreflexive

and symmetric) respectively. Two events which are neither in conflict nor related by causality are said to be concurrent. If there are no concurrent events, the structure is called sequential. In order to be able to use event structures as a model for our objects, we have to attach to them some more information in the form of event labels. We introduce a labelling function $\mu : Ev \rightarrow \mathcal{P}(Ac)$, i. e. a total function attaching to each event the actions which occurrence it denotes. A prime event structure enriched by a labelling function is called a labelled prime event structure. Labelled prime event structures constitute a non-interleaving model of concurrency and are a fair choice when a denotational semantics is needed.
Objects in TROLL do not have intra-object concurrency, and are therefore modelled by sequential labelled event structures. Let (E_i, μ_i) be a model for the object instance $i \in Id$. A system model is obtained by composing the corresponding instance models. The idea of the composition construction is to identify shared communication events and leave all the other events concurrent. Fig. 3.8 shows part of the CATC system model focusing on the communication between a staff (`Users(1)`) and an operator (`Users(2)`) instances. We represent events by boxes with the corresponding label (set of

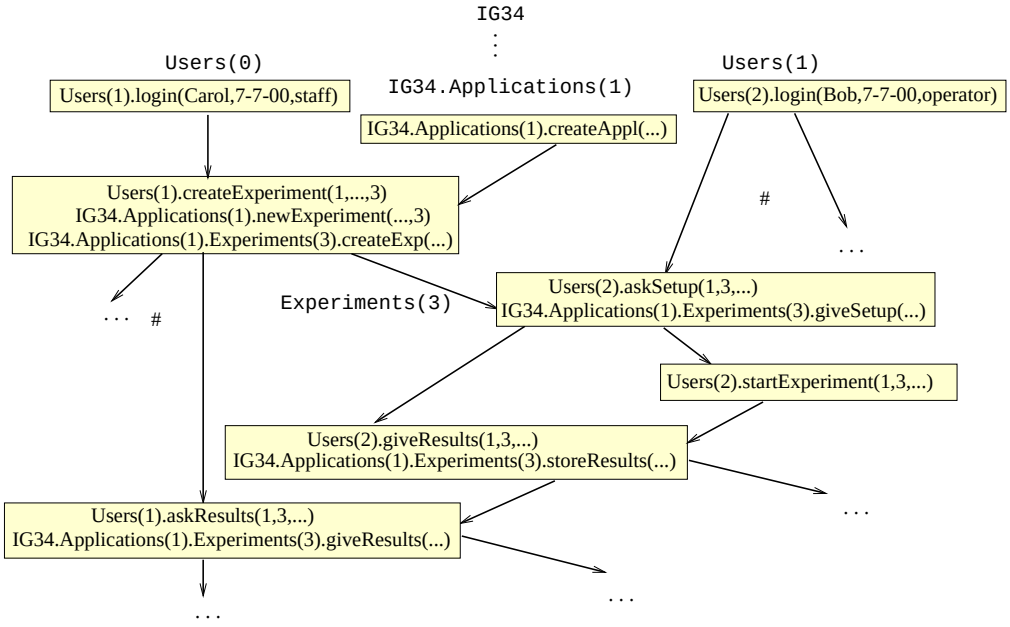


Figure 3.8: Extract from the CATC System Model

actions) written inside. Arrows between boxes denote the causality relation among events. The communication, as presented already in the OMTROLL communication diagram of Fig. 3.6, is done indirectly via experiments.

3.5 Summary

In this chapter, we have introduced the modelling with OMTROLL and TROLL by example. OMTROLL consists of several diagrams that allow developers to give a first overview of the system. They are summarised as follows:

- The *community diagram* gives an overview of the object classes and their respective relationships to each other.
- The *data type diagram* represents user defined data types.
- The *object class declaration diagram* describes the attributes and actions of each object class.
- The *behaviour diagram* defines the life cycle of the objects.
- The *communication diagram* establishes the communication structure between the objects of the system.

A frame of a TROLL specification can be derived from the OMTROLL diagrams. This has to be further refined to a complete system specification. A TROLL specification consists of the following parts:

- *Data Type Definitions:* This part includes the definition of user data types.
- *Object Class Specifications:* They define prototypical object descriptions. An object class consists of a signature and a behaviour definition. Attributes and actions are declared in the signature part. The behaviour of an object is defined by the description of rules for its actions and the declaration of integrity constraints. Object classes may be complex. They may have other object classes as components as well as specialisations.
- *Object Declarations:* Objects are declared over object classes. These declarations describe the set of concurrent objects in the system.

- *Global Behaviour Specification:* This part describes the interactions between concurrent objects in the system. All actions involved in an interaction take place simultaneously.

We have described briefly the formal semantics underlying the TROLL language. A TROLL specification is formalised by a system signature and a set of axioms in logical formulae. The system signature describes the structural part of the specification. The set of axioms describe the system dynamics. Axioms are expressed in a distributed temporal logic and interpreted over labelled prime event structures.

In the next chapters, we present the steps necessary for developing a software environment that assists developers in the modelling and validation through animation of TROLL specifications.

Chapter 4

Analysis

In the TROLL development environment, specifications are firstly analysed and stored into an intermediate representation. Several checks assure that the specification is correct with respect to the syntax and static semantics of the specification language. During these checks an internal representation of the specification is created. The internal representation structures the specification in such a way that makes easier the development of subsequent tools such as interpreters and code generators. These activities are similar to those done in the analysis phase of a compiler. In this chapter, after a brief introduction to the analysis phase, we present the internal structure that is used for storing a TROLL specification. We then describe which rules have to be checked in the semantic analysis of TROLL specifications.

4.1 Introduction

Some parts in the construction of an executable prototype from a TROLL specification are similar to the traditional construction phases of a compiler. The tasks of a compiler are usually structured in two main phases: *analysis* and *synthesis* [ASU86]. The analysis phase splits the source code into constituent items and creates an internal representation of the source code. The synthesis phase generates the target code from the internal representation. The synthesis phase will be presented in the next chapters. Fig. 4.1 shows the activities to be done in the analysis of TROLL specifications. These activities are:

- *Lexical Analysis (scanning):* In this phase, the stream of characters

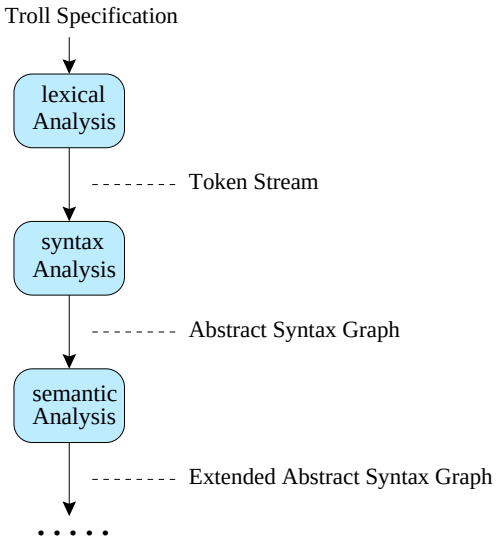


Figure 4.1: Analysis Phase of TROLL Specifications

building a TROLL specification is read and structured into lexical symbols also called *tokens*. Lexical symbols are, for instance, reserved words, identifiers and constants. Symbols which are not significant for subsequent phases such as blanks and comments are ignored.

- *Syntax Analysis (parsing):* The syntax analysis checks the syntax of the specification. It groups the tokens from the lexical analysis into grammatical phrases. The syntactic structure of a TROLL specification is described by a context free grammar (see Appendix A.2). The syntax analysis creates an abstract syntax graph from the specification. This graph is a structured representation of the specification and is used as input of the subsequent phases.

- *Semantic Analysis:* The semantic analysis checks the static semantics of the specification. Semantic properties to be analysed in this phase are those which depend neither on objects' states nor on input values, i. e. they are the same in each possible execution. Examples of static checks are type checks and uniqueness checks. As a result of this phase, the syntax graph is extended with new vertexes and edges that represent semantic relationships between different parts of the specification. These extensions facilitate the work in the next phases such as

the generation of code.

Errors in the specification are handled and reported by each of these phases. Lexical errors are characters in the specification that do not match any word (token) of the language. The syntax analysis detects errors where the token sequence does not correspond to any well-formed phrase of the grammar. Constructs that have a correct syntax structure but not correct meaning are detected by the semantic analysis.

Often, the analysis phase of a compiler is called the *front end* whereas the synthesis phase is called the *back end*. The front end represents the parts of a compiler which depend exclusively on the source language and are independent of the target machine. On the other hand, the back end consists of those parts which depend on the target machine. In the TROLL "compiler" we can also do this distinction. The analysis phase checks a specification and creates an internal representation independently of the target programming language. Moreover, the analysis phase is also independent of the tools that will be used next. We could not only develop code generators but also other tools such as interpreters, pretty printers and documentation tools.

In the next section, we present the structure of the syntax graph. We show then the list of static rules that can be checked in the semantic analysis. The analysis phase will be further explained when describing its implementation in Sect. 7.2.4.

4.2 Abstract Syntax Graph

As depicted in Fig. 4.1, an abstract syntax graph is used as the internal data structure of a TROLL specification. This structure stores not only context-free but also context-sensitive information about the specification. Since context-sensitive relationships are represented by edges between related nodes, it is possible that a node has more than one input edge. So a graph and not a tree is used as data structure. The syntax graph is defined as a directed attributed labelled graph. Its structure is based on [Eng86]. Fig 4.2 shows the OMTROLL community diagram of the syntax graph. A graph has an anchor (or root) vertex that represents the start symbol of the TROLL grammar (<systemSpec>). The graph is composed of vertexes and edges. Vertexes and edges are identified by a number (`vertex(id)` and `edge(id)` respectively). The attributes `next_vertex_id` and `next_edge_id` of the graph represent the identities of the next vertex and the next edge to

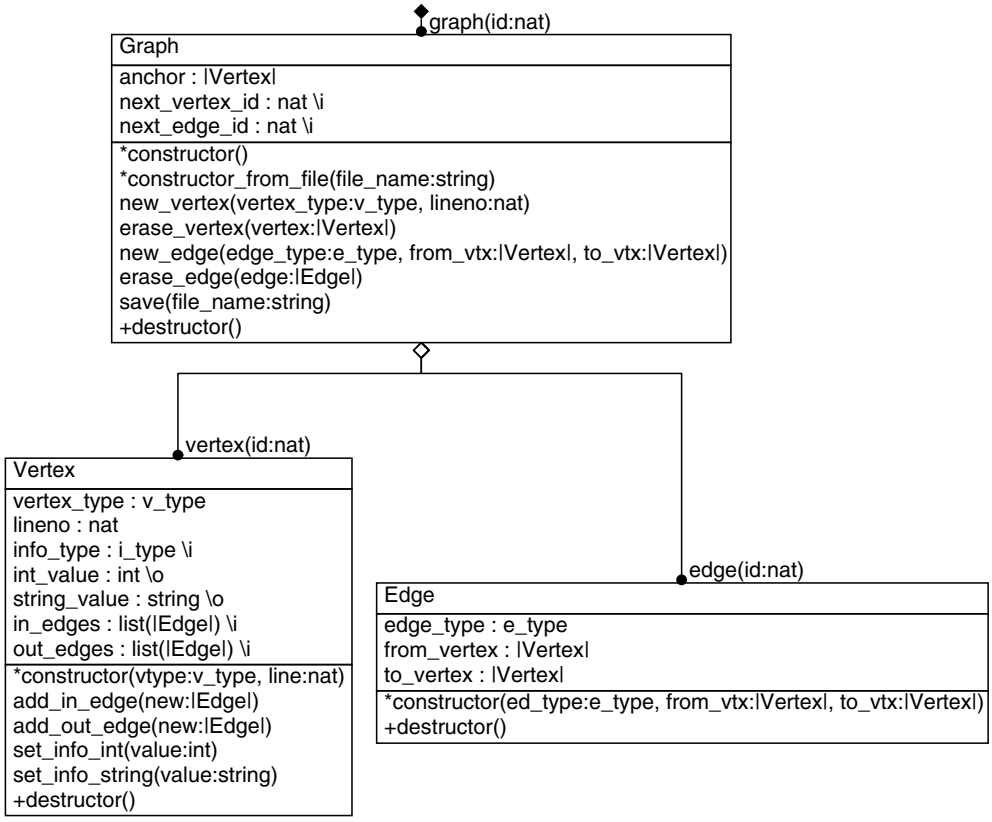


Figure 4.2: OmTroll Community Diagram of the Syntax Graph

be created respectively. Vertexes are labelled by the attribute `vertex_type`. The label of a vertex is derived from the grammar symbol represented by the vertex preceded by the prefix `"V_"`. For instance, a vertex corresponding to the grammar symbol <dataTypeSpec> is labelled by `V_dataTypeSpec`. Vertexes are connected through directed edges. Each vertex has a list of input edges (`in_edges`) as well as a list of output edges (`out_edges`). Vertexes are attributed. Besides information about its label and input/output edges, a vertex stores the following information:

- The line number of the Troll specification where the token represented by the vertex is defined (`lineno`). This information is used whenever an error is found to report its localisation in the source code.

- A value whose meaning depends on the kind of token represented by the vertex. Values can represent, for instance, the name of an object class, a constant or whether an action is a birth action. The data type of this value is given by the attribute `info_type`. Values of type integer are stored in the attribute `int_value` and values of type string are stored in the attribute `string_value`.

Edges are labelled by the attribute `edge_type`. The label of an edge is derived from its terminal vertex and is prefixed by `"E_"`. For example, an edge with its terminal vertex labelled by `V_ident` (which represents an <identifier>) is labelled by `E_ident`. An edge connects an initial vertex (`from_vertex`) to a terminal vertex (`to_vertex`).

The graph is generated by the parser that extends it every time a derivation rule of the context free grammar is used. Thus the structure of the graph is based on the productions of the grammar applied in the derivation process. Some particularities of the graph are:

- Derivations where the grammar rule has only terminals at its right side are represented by a unique vertex. The vertex is labelled by the nonterminal at the left side of the rule. The corresponding terminal at the right side of the rule is stored in an attribute. For instance, Fig. 4.3 shows the representation of a boolean operator in the graph. The boolean operator is represented by a vertex `V_boolOp` which contains the operator type in the attribute `int_value`.

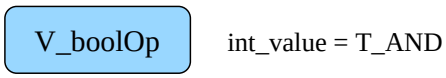


Figure 4.3: Representation of a Boolean Operator in the Graph

- Derivations where the right side of the grammar rule describes a list are represented by a root vertex labelled by the nonterminal at the left side of the rule. The root vertex has as children the elements of the list. The order in the list is given by additional edges. The first and last element in the list are denoted by the edges `E_first` and `E_last` going from the root vertex to the respective elements. The order among the elements is given by `E_next` edges. As an example, consider the grammar production associated to action rules in TROLL:

<actionRule_list> ::= <actionRule> { “,” <actionRule> }

When applying this rule the parser creates a vertex `V_actionRule_list` and hangs on it the subgraphs corresponding to each action rule. This is illustrated in Fig. 4.4.

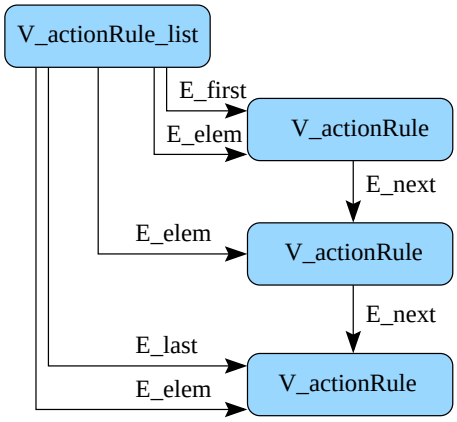


Figure 4.4: Representation of a List of Action Rules in the Graph

The graph represents the abstract syntax of the specification. That is, information about the concrete syntax, unimportant for a translation, does not appear in the graph. So syntactic constructs such as reserved words, brackets and delimiters are not stored. This makes graphs smaller and easier to manage in the next phases. Redundancies are also eliminated. Intermediate vertexes, i. e. vertexes with only a child and whose meaning is just to connect the upper vertex with the child vertex not giving any extra information at all, are removed from the graph. For instance, a direct representation of a conditional term would require an intermediate vertex denoting that the conditional term is a data term and a child vertex describing its specialisation as conditional. In this case, the graph is simplified by deleting the intermediate vertex. Fig. 4.5 shows this case. The representation of data terms expressing operations can also be simplified. In this case and unlike the example above, the intermediate vertex denoting that the expression is a data term has more than one child: one for the operator type and so many as operands. The simplification consists in replacing the intermediate vertex with the vertex

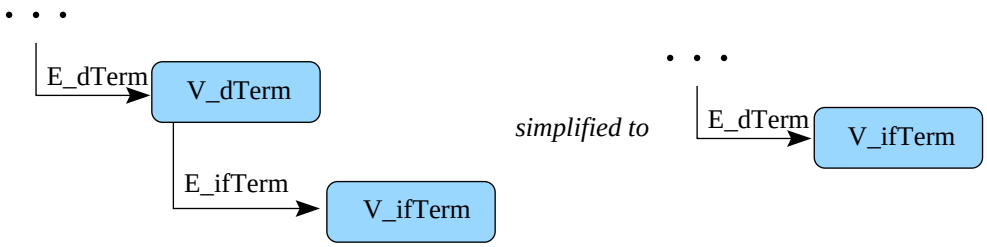


Figure 4.5: Simplification of Vertexes representing Conditionals

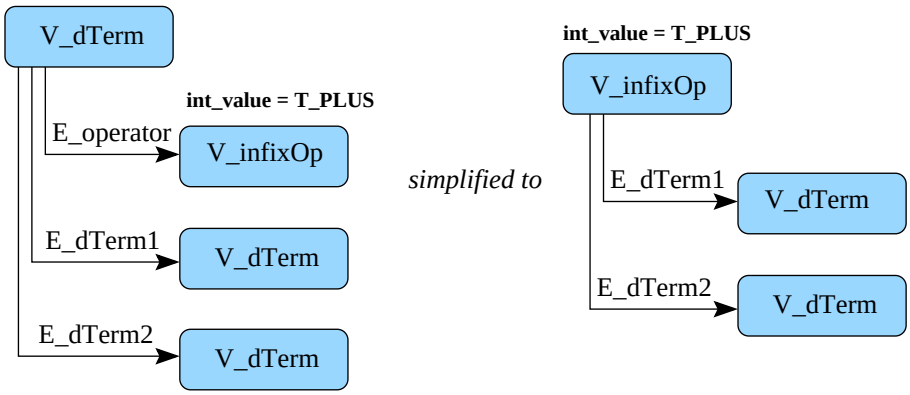


Figure 4.6: Simplification of Vertexes representing Operations

representing the operator type. For example, the representation of a data term describing an addition is depicted in Fig. 4.6.
The reverse generation of the original text specification from the syntax graph requires a transformation process called *unparsing*. Since the graph represents the abstract syntax of the specification, missing information about the concrete syntax must be introduced by the unparser. The documentation tool of the workbench, *trldoc*, makes an unparsing from the graph and generates HTML code which documents the specification. *trldoc* will be presented in Sect. 7.2.5.

Besides checking the static semantics of the specification, the semantic analyser adds new information to the graph. This information is context-sensitive and is not given by the *EBNF* syntax of the language. It includes, for instance, the data type of each data term and the declaration place of each identifier. Subsequent phases make use of this information. For example, the

code generator needs to know whether an identifier represents an attribute, an argument or a local variable in order to generate the correct code. In the TROLL environment, all data concerning a specification are directly stored in the graph, i. e. additional data structures such as symbol tables are not managed. This makes the graph the only data structure to be managed by the tools. A possible way of storing context-sensitive information in the graph consists in introducing the information in attributes defined in the vertexes. For instance, the data type of a data term could be stored in a `type` attribute defined in the vertex denoting the data term. Since not all vertexes require the same kind of attributes, a problem with this technique is that vertexes may not have a uniform structure. It depends on the kind of information to be stored in the vertex. On the other hand, the definition of a uniform structure for all vertexes would mean that some vertexes would contain unnecessary attributes. In both cases, extensions of the required attributes would entail changes in the structure of the vertexes as well as new access functions. A better and more natural way of storing context-sensitive information in the graph consists in representing the information by extending the graph with new vertexes and edges. This is the technique used in the TROLL environment. For example, the data type of a data term is represented by an `E_type` edge which relates the vertex denoting the data term to the vertex denoting the data type definition. With this technique, the structure of the vertexes remains the same independently of the information to be stored.
The structure of the abstract syntax graph is presented in Appendix C. To make easier the access to the graph, an interface has been implemented. The interface hides details from the graph structure and provides the tools with access functions such as retrieving the name of all object classes in the specification or the data type of action parameters. The syntax graph interface is described in [Voe99]. In the next section, we describe which kind of semantic rules are checked in the semantic analysis.

4.3 Semantic Analysis

Once the syntax of a TROLL specification has been checked by the parser and before the animation can begin, the *well-formedness* of the specification has to be analysed. The well-formedness of a specification is given by its static semantics. Static semantics have nothing to do with meaning. They

describe the relations between the signs of the language that are not directly expressed by the EBNF syntax and can be checked at analysis time. For programming languages, a static semantics checker is normally referred to as type checker. Rules to be checked by the TROLL semantic checker include:

- *Uniqueness Rules:* Every declared identifier should be unique.
- *Identification of Identifiers:* Every referenced identifier must be declared. Identifiers have to be *identified* in the sense that each identifier has to be associated to its respective declaration.
- *Type Consistency Rules:* Data terms included in an expression have to be checked for type consistency.
- *Other Rules:* These are rules which are not contained in any of the previous groups. For instance, a class cannot be a component class of itself.

Other semantic rules can be checked only dynamically. For example, divisions by zero and range overflows that depend on input values can only be checked on run time. Well-formed TROLL specifications are those having at least one model in the dynamic semantics. In general, it is not statically decidable whether a given TROLL specification is well-formed or not. As an example, consider the following fragment of a TROLL specification corresponding to the behaviour specification of an action:

```
...
a := 10,
...
if x = true
  then a := 15
fi,
...
```

We can see that the specification is not well-formed if `x` is equal to `true` because then two different values would be assigned to `a` in a state transition. Since conditions may be arbitrarily complex, this cannot in general be checked statically. We will come back to this issue in Chapter 6.

Attribute grammars are normally used for specifying the static semantics of languages. An attribute grammar extends a context free grammar with semantic information by attaching attributes to the grammar symbols. Values

for attributes are computed by semantic rules associated with the grammar productions. Attribute grammars are specially appropriate for automated compiler generation. Since that is out of the scope of the present work, we do not use attribute grammars for describing the static semantics of TROLL specifications, but just define the semantic rules to be checked by the semantic analyser. In the next, we show these rules. Because checks of uniqueness, declaration of identifiers and type consistency are similar to those of standard programming languages, we will describe them only briefly and concentrate on checking those rules which are special for TROLL.

4.3.1 Uniqueness Check

Uniqueness check assures that each identifier is declared uniquely. That is, the declaration of each identifier in the specification is unambiguously identified. The uniqueness rules in TROLL can be summarised as follows:

- The identifiers inside each declaration group (data types, object classes and objects) must be unique.
- The names of elements inside the signature declaration of an object class (components, attributes and actions) must also be distinct. These names are not in conflict with names used in the signature of components because elements of components are unambiguously identified by *dot notation* (an element of a component is referenced by the component's identifier, followed by a dot and the element's name). On the other hand, since the signature of a specialised class includes the signature of its superclass, names of components, attributes and actions in the specialisation must also be distinct to those defined in the superclass.
- The parameter names of an action declaration must be pairwise disjoint.
- The behaviour specification of an action inside a class may be defined only once.
- Variables declared inside the behaviour specification of an action must be unique. Variables and action parameters cannot have the same names. On the other hand, variables and attributes may have the

same names. In this case, the attribute is shadowed by the variable. The scope and visibility of identifiers determine which declaration is associated to each referenced identifier in the specification. We define how declarations are assigned to identifiers in the next section.

- The elements of an enumeration type as well as the field names of a record type have to be pairwise disjoint.

4.3.2 Identification of Identifiers

The semantic analyser must check that each identifier in the specification has been declared. For this task, the analyser looks in the specification for the respective declarations. The search strategy depends on the kind of declaration and the scope and visibility rules. In TROLL, data type and class names are always visible, and the search takes place in the corresponding declaration sections. For variables, action parameters and attributes, the search sequence is defined as follows:

1. If the identifier belongs to an expression which is in the scope of a range declaration, the analyser looks first at all if the identifier was implicitly declared in the range declaration. Expressions that define range declarations are, for instance, quantified formulas:

   ```
   (all x in S) (x > 10)
   ```

2. If the identifier is inside the behaviour specification of an action, the analyser looks if the identifier is declared as a local variable.

3. In case the identifier is inside the behaviour specification of an action, the analyser looks if the identifier is declared as an action parameter.

4. The analyser looks if the identifier is declared as an attribute in the class.

5. Finally, and in case the identifier is used in a specialised class, the analyser looks if the identifier is declared as an attribute in the superclass. Since the superclass may also be the specialisation of another class, the search is defined recursively.

If the declaration of the identifier is not found, the checker reports an error message.

4.3.3 Type Check

Type checking is the main task of static analysers. A type checker verifies that the type of each data term in the specification is correct according to the context. TROLL is strongly typed, i. e. each variable must be declared before it is used. The type of a variable is determined in its declaration and never changes. This makes possible the detection of type errors statically. Nevertheless, there are some checks that can be performed only dynamically. Examples of dynamic checks are divisions by zero, range overflows in the selection of list elements and the check whether the substraction of two natural numbers is also a natural number.
Expressions that can be statically type checked are:

- Data type specifications cannot be defined recursively, i. e. given a data type specification *data type* <ident> = <type>, <type> cannot depend on <ident> neither directly nor indirectly.
- Operands in data terms must have compatible types and the operator must be defined for the type of the operands.
- The parameter type of multiple objects and components must be *nat*, *char*, *string* or *enum*.
- The type of an initialised attribute must be compatible with the type of the initialisation value.
- A derived attribute and its derived term must have compatible types.
- The type of test expressions in conditionals must be boolean.
- The left and right parts of assignments must have compatible types.
- The type of action parameters must be compatible with the types defined in the action declaration.

For a more complete description of the type checker see [Rüt99].

4.3.4 Other Rules

Here we list the semantic rules that are not contained directly in the above groups. They include mostly visibility and restriction rules. Before listing

the rules, we give some definitions that will help to simplify the presentation. We start with the definition of visibility in an expression.

Definition 4.3.1 Let *expr* be an expression, its visibility, $vis(expr)$, is defined as the set of attributes, parameter variables and local variables which may appear in the expression. □

We will use this definition in the presentation of the visibility rules. Next, we classify attributes and actions according to their visibility inside the classes.

Definition 4.3.2 Let C be a class, we define:

- $Attr_{read}(C)$ as the set of attributes whose values can be read in C
- $Attr_{write}(C)$ as the set of attributes whose values can be set in C
- $Act(C)$ as the set of actions that can be called by actions of C and whose behaviour can be defined or extended in C.

$Attr_{read}(C)$, $Attr_{write}(C)$ and $Act(C)$ are constructed as follows:

1. Let $Attr_{loc}(C)$ and $Act_{loc}(C)$ be the set of attributes and actions respectively which are declared in the local signature of C then
   - $Attr_{loc}(C) \subseteq Attr_{read}(C)$
   - Let $Attr_{der}(C) \subseteq Attr_{loc}(C)$ be the set of attributes declared as *derived* in C then $Attr_{loc}(C)$ - $Attr_{der}(C) \subseteq Attr_{write}(C)$
   - $Act_{loc}(C) \subseteq Act(C)$
2. If C has components and let C' be a component class or a subclass of a component class then
   - Let $Attr_{hidden}(C') \subseteq Attr_{loc}(C')$ be the set of attributes declared as *hidden* in C' then $Attr_{read}(C')$ - $Attr_{hidden}(C')$ - $Attr_{read}(C'') \subseteq Attr_{read}(C)$ for each C'' hidden component of C'.
   - Let $Act_{hidden}(C') \subseteq Act_{loc}(C')$ be the set of actions declared as *hidden* in C' then $Act(C')$ - $Act_{hidden}(C')$ - $Act(C'') \subseteq Act(C)$ for each C'' hidden component of C'.
3. If C is declared as a specialisation and let $supC$ be its superclass then

- $Attr_{read}(supC) \subseteq Attr_{read}(C)$
- $Attr_{write}(supC) \subseteq Attr_{write}(C)$
- $Act(supC) \subseteq Act(C)$

4. Nothing else belongs to $Attr_{read}(C)$, $Attr_{write}(C)$ and $Act(C)$

□

The local signature of a class is always visible inside the class. Derived attributes can be read but not written. The signature of a class is extended by the visible signature of its components, i. e. attributes and actions of the components that are not declared as hidden are embedded in the compound class. To restrict the visibility of a component to the class where it is declared, the component may be declared as hidden. This means that the component is only visible in the class where it is declared but not in upper classes. So if a class C declares a component C' as hidden, and if C is in turn a component of a class C'', then the signature of C' is not visible in C''. A component may also have some specialisation aspects. In order to refer to attributes and actions declared in specialisations, the visible signatures of classes that are specialisations of components are also embedded in the composite. Since a component gets its specialisation aspects in the moment of its birth, it is statically undecidable to know whether a component has some specialisation aspects or not. This means that references to attributes and actions of non-existent specialisation aspects must be checked dynamically. Nevertheless, the *isA* operator can be used to find out whether a specialisation aspect is valid before referencing to attributes and actions of the specialisation. We illustrate the use of the *isA* operator by example.

Example 4.3.1 *(isA Operator)* Given the following TROLL specification:

```
object classA
 components Cs(n:nat):C;
 ...
 behavior
  actA(n)
  do
   if Cs(n) isA(S)
   then S(Cs(n)).actS
   fi
   ...
  od;
 ...
end;
```

```
object class C
 ...
end;

object class S
 aspect of C if ...
 ...
 behavior
  actS
  do...od;
 ...
end;
```

Objects of class A may have components of class C that may have a specialisation S. The occurrence of actA, defined in the class A, entails the occurrence of actS, defined in the specialisation class S, only if the component Cs(n) has the aspect S. If actS is directly called without previous use of the operator *isA* then an error is produced on run time in case that Cs(n) has not the aspect S. □

Compound classes have read-access but not write-access to the attributes of the components. An attribute can only be written in the class where it is declared or in specialisations of the class. So $Attr_{write}$ does not include attributes of components. Components have not access to the signature of the compound class nor to the signature of other components of the compound class. If a component wants to interact with other components, this can be done by extending the behaviour of its actions in the compound class. We will show an example of this later.

A specialisation class has access to the attributes and actions of its superclass. Unlike the embedding of components in the compound classes, the hidden attributes and actions of the superclass are visible in the specialisation. Moreover, specialisations have write access to the attributes of the superclasses. Thus attributes and actions of a superclass may be used in the subclasses in the same way as in the superclass.

In the next, we present semantic rules that are statically checked. Rules are specified in a syntax-directed manner, so the reader can easily find the concepts introduced in the previous chapter. We start with the declaration of specialisations.

Specialisation Declaration

Rules to be checked in a specialisation declaration are those concerned with the validity of the superclass, the action that determines the specialisation and the visibility of specialisation formulas.

Rule 4.3.1 Given a specialisation declaration $SpecDecl = (SupC, SpecCondList)$ declared in a class C, where $SupC$ is the superclass identifier and $SpecCondList$ is the list of specialisation conditions, then

- $SupC$ must be a class identifier different to C and cannot have any component or specialisation relationship with C.
- Let $SpecCond = (Act, ParamList, Form)$ be a specialisation condition in $SpecCondList$ where Act is the specialisation action, $ParamList$ a list of parameter identifiers and $Form$ is an optional formula then
  - Act must be a birth action of $SupC$
  - $ParamList$ must have the same number of elements that the number of parameters defined in the declaration of Act
  - $vis(Form) \subseteq ParamList$, where $vis(Form)$ represents the visibility of $Form$ as defined in Def. 4.3.1.

□

The superclass cannot be a specialisation nor a component of its specialisations. In this way, cycles in the class hierarchy are avoided. No attributes can appear in a specialisation formula. That is due to the fact that an object gets its specialisation when it is born. So its attributes cannot still be read because they do not have any value in the previous state.

Component Declaration

As in a specialisation declaration, it has to be checked that there are not any cycles in the component relationships. Therefore a component class cannot be a component of itself, nor a compound class nor a specialisation of the class where it is declared.

Rule 4.3.2 Given a component declaration $CompDecl = (Id, Param, CompC)$ declared in a class C where Id is the component name, $Param$ an optional parameter identifier and $CompC$ the component class then

- $CompC$ must be a class identifier different to C and cannot have any component or specialisation relationship with C.

□

Attribute Declaration

Checks in the attribute declaration of a class are those that assure a valid combination of attribute features as well as a valid derivation term for derived attributes.

Rule 4.3.3 Given an attribute declaration $AttrDecl = (Id, Type, Desc)$ declared in a class C where Id is the attribute name, $Type$ is the attribute type and $Desc$ is the list of attribute features then

- The following non-meaningful combination of attribute features are not allowed in $Desc$:

derived	-	constant
derived	-	optional
derived	-	initialized
optional	-	constant
optional	-	initialized

- if the attribute is a derived attribute and $Dterm_{der}$ is its derivation term then $vis(Dterm_{der}) \subseteq Attr_{read}(C)$ - $Attr_{dep}(Id)$ where $Attr_{dep}(Id)$ is the set of derived attributes whose derivation terms depend directly or indirectly on Id.

□

With the exception of *hidden*, it is not allowed to specify further features for derived attributes. Extra features are given by the features of the attributes on which the derivation term depends. An explicit declaration of a derived attribute as constant, optional or initialised would lead to an overspecification

when not to inconsistencies. The specification of an attribute as optional and constant or as optional and initialised is a contradiction. It has also to be checked that the derivation term of a derived attribute does not depend on the attribute itself.

Action Declaration

Restrictions in the specification of action declarations are given by the next rule.

Rule 4.3.4 Given the list of action declarations $ActDeclList$ defined in a class C then

- If C is not declared as a specialisation then $ActDeclList$ must contain at least one birth action.
- If C is declared as a specialisation then $ActDeclList$ must not contain any birth or death actions.
- Let A be a a birth action declared in $ActDeclList$ then A cannot be hidden.

□

Each object class with the exception of specialisation classes must have at least a birth action declaration because objects can only be created by the occurrence of birth actions. A specialisation class may not have neither birth nor death action declarations because they are inherited from its superclass. Moreover, a birth action cannot be hidden. The declaration of a birth action as hidden would mean that the action can only be called internally. This is not possible because the first action to be executed in the object's life is always the birth action.

Behaviour Specification

The behaviour specification of an action entails the definition of action terms, preconditions, local variables and action rules. In the next, we discuss checks to be done in each of these definitions.

Action Definition

The set of actions whose behaviour may be defined/extended in a class is defined according to Def. 4.3.2.

Rule 4.3.5 Given an action definition $ActDef = (A, ParamList)$ defined in a class C where A is an action term and $ParamList$ is an optional list of parameter identifiers then

- $A \subseteq Act(C)$
- $ParamList$ must have the same number of elements that the number of parameters defined in the declaration of A.

□

A is an action term and not just an identifier because as mentioned previously, classes may extend the behaviour of actions declared in its components. Next example illustrates this case.

Example 4.3.2 *(Extension of component actions)* Given the following TROLL specification:

```
object classA
 components Cs(n:nat):C;
 ...
 behavior
  Cs(n).actC(p1)
  do
   ...
  od;
 ...
end;
```

```
object class C
 ...
 actions
  actC(p1:nat);
 ...
end;
```

The behaviour of `actC` declared in the class `C` is extended in the compound class `A`. □

As can be seen in the example above, the definition of an action can introduce new variables (in the example, `n` and `p1`). They can be used in the behaviour specification of the action. The following definition classifies these variables.

Definition 4.3.3 Given an action definition $ActDef = (A, ParamList)$ defined in a class C where A is an action term and $ParamList$ is an optional list of parameter identifiers, then we define

- $Var_{id}(A)$ as the set of variables implicitly declared in the identification of A.
- $Param_{in}(A) \subseteq ParamList$ as the set of input parameters.
- $Param_{out}(A) \subseteq ParamList$ as the set of output parameters.

□

Preconditions

The visibility of an action precondition is defined by the next rule.

Rule 4.3.6 Given a precondition $Prec$ for an action A defined in a class C then

- If A is a birth action of class C, then $vis(Prec) \subseteq Param_{in}(A) \cup Var_{id}(A)$ else $vis(Prec) \subseteq Attr_{read}(C) \cup Param_{in}(A) \cup Var_{id}(A)$

□

As in specialisation formulas, no attributes may appear in the precondition of a birth action because they do not have any value in the previous state. Output parameters may not be used in a precondition because its value will be set in the current state transition and do not have any value before.

Before the definition of rules to describe the behaviour of actions, local variables may be declared. We introduce them in the next definition.

Definition 4.3.4 Let A be an action defined in a class C then $Var_{loc}(A)$ is defined as the set of local variables declared in the behaviour specification of A. □

Action Rules

Action rules determine the effect of an action on the objects' states by means of valuations and calling rules.

Valuations

In valuation rules, we distinguish between variables whose value can be assigned and variables whose value can be read.

Rule 4.3.7 Given a valuation rule $Val = (AssigTerm, DTerm)$ defined in an action A of a class C where $AssigTerm$ is the assign term at the left side and $DTerm$ is the data term at the right side of the rule, then

- $vis(AssigTerm)$ is defined as follows:
  - if A is declared as a birth action then $vis(AssigTerm) \subseteq Attr_{write}(C)$ - $Attr_{init}(C) \cup Param_{out}(A) \cup Var_{loc}(A)$, where $Attr_{init}(C)$ is the set of attributes declared as initialised.
  - if A is declared as a death action then $vis(AssigTerm) \subseteq Param_{out}(A) \cup Var_{loc}(A)$
  - Otherwise, $vis(AssigTerm) \subseteq Attr_{write}(C)$ - $Attr_{const}(C) \cup Param_{out}(A) \cup Var_{loc}(A)$, where $Attr_{const}(C)$ is the set of attributes declared as constant.
- $vis(DTerm)$ is defined as follows
  - if A is declared as a birth action then $vis(DTerm) \subseteq Param_{in}(A) \cup Var_{id}(A) \cup Var_{loc}(A)$
  - Otherwise, $vis(DTerm) \subseteq Attr_{read}(C) \cup Param_{in}(A) \cup Var_{id}(A) \cup Var_{loc}(A)$

□

The left side of an assignment represents the value of the variables in the next state whereas the right side represents the value in the current state. Constant attributes can only be assigned by birth actions in case they are not initialised. Death actions cannot change the value of attributes. Assignments to input parameters are not allowed. Attribute values can only be observed for living objects. Thus attributes can not be observed during the occurrence of a birth action.

Calling Rules

In a calling rule, the semantic checker has to assure that the action to be called is visible in the class where the rule is defined and that data terms representing action parameters are valid.

Rule 4.3.8 Given a calling rule $CallRule = (A', DTermList)$ defined in an action A of a class C where A' is an action term and $DTermList$ is a list of data terms then

- $A' \subseteq Act(C)$ and $vis(A') \subseteq Param_{in}(A) \cup Var_{id}(A) \cup Var_{loc}(A)$ in case A is a birth action, otherwise $vis(A') \subseteq Attr_{read}(C) \cup Param_{in}(A) \cup Var_{id}(A) \cup Var_{loc}(A)$
- $DTermList$ must have the same number of elements that the number of parameters defined in the declaration of A'
- Let $DTerm_{in}$ and $DTerm_{out}$ be data terms of $DTermList$ that refer to an input parameter and an output parameter respectively of action A' then
  - $vis(DTerm_{in})$ is the same as the visibility of data terms at the right side of a valuation (see Rule 7).
  - $vis(DTerm_{out})$ is the same as the visibility of assign terms at the left side of a valuation (see Rule 7).
- if A is a birth action of C then A' cannot be a death action of C and vice versa.

□

As in the definition of an action (see Rule 5), A' may be an action of a component and may be therefore parametrised in order to identify the component object. So its visibility must be defined. The visibility rules of data terms representing action parameters correspond to those defined for assignments. An object cannot be born and dead in the same state transition. Since action calling is transitive and since the occurrence of an action may be conditioned, the last restriction can only be partially checked in analysis time. That is, it could be possible that a birth action calls an update action which in turn calls a death action. Although in some cases, it could be statically possible to determine the set of actions to take place in a state transition, in general,

this can only be determined on run time. So the checker can only detect the occurrence of birth and death actions in the same state transition if the birth action calls directly a death action or vice versa.

Integrity Constraints

Integrity constraints restrict the values of the attributes in the classes. The visibility in the definition of an integrity constraint is given by the next rule.

Rule 4.3.9 Given an integrity constraint $Constr$ defined in a class C then

- $vis(Constr) \subseteq Attr_{read}(C)$

□

Global Behaviour Specification

The global behaviour of an action, i. e. the specification of its interactions with actions belonging to other objects, is defined in the global behaviour specification part. In the previous chapter, example 3.3.6 illustrated a global behaviour specification. Rules to be checked in the global specification of an action are similar to those defined for the local behaviour specification.

Rule 4.3.10 Given a global action definition $ActDef = (A, ParamList)$ where A is an action term and $ParamList$ is an optional list of parameter identifiers then

- $ParamList$ must have the same number of elements that the number of parameters defined in the declaration of A.
- Let $Prec$ be a precondition defined in the specification of A, then $vis(Prec) \subseteq Param_{in}(A) \cup Var_{id}(A)$
- Let $Val = (AssigTerm, DTerm)$ be a valuation rule defined in A where $AssigTerm$ is the assign term at the left side and $DTerm$ is the data term at the right side of the rule then
  - $vis(AssigTerm) \subseteq Param_{out}(A) \cup Var_{loc}(A)$
  - $vis(DTerm) \subseteq Param_{in}(A) \cup Var_{id}(A) \cup Var_{loc}(A)$
- Let $CallRule = (A', DTermList)$ be a calling rule defined in A where A' is an action term and $DTermList$ is a list of data terms then

- A' must not be declared as hidden.
- $vis(A') \subseteq Param_{in}(A) \cup Var_{id}(A) \cup Var_{loc}(A)$
- $DTermList$ must have the same number of elements that the number of parameters defined in the declaration of A'
- Let $DTerm_{in}$ and $DTerm_{out}$ be data terms of $DTermList$ that refer to an input parameter and an output parameter respectively in action A' then
  - $vis(DTerm_{in}) \subseteq Param_{in}(A) \cup Var_{id}(A) \cup Var_{loc}(A)$
  - $vis(DTerm_{out}) \subseteq Param_{out}(A) \cup Var_{loc}(A)$

□

Unlike in the local behaviour specification part, there is not restriction in the set of actions whose global behaviour can be specified. All actions may have a global behaviour. Moreover, since interactions take place between objects of different hierarchies, any action may be called with the exception of actions declared as hidden. In the global behaviour specification, actions have not access to attributes.

In this section, we have analysed semantic rules that can be statically checked. As we have mentioned, there are still checks that can only be done dynamically. These checks will be discussed in the next chapters.

4.4 Summary

In this chapter, we have presented the analysis phase required for constructing an animator of TROLL specifications. This phase has two purposes. On the one hand, several checks assure that the specification is correct with respect to the syntax and static semantics of the TROLL language. On the other hand, an internal data structure is created from the specification. This structure is used as a central repository for the remaining phases. In this chapter, we have particularly discussed how TROLL specifications are represented by the internal data structure and which semantic rules can be checked statically.

A TROLL specification is internally represented as a directed attributed labelled graph. It is generated by the parser and represents the abstract syntax of the specification. The semantic analyser extends the graph with

context-sensitive information that is necessary in the subsequent phases. The particularities of the graph are:

- The graph stores only the abstract syntax of the specification. In this way, the graph is smaller and easier to manage in the next phases.
- Moreover, the graph is simplified by eliminating unnecessary vertexes.
- The technique used for extending the graph with context-sensitive information consists in introducing new vertexes and edges in the graph. The structure of the graph remains the same independently of changes in the required information.
- The graph is the only structure used for storing the specification. All the remaining tools access to the graph by the same interface.

We have presented static rules to be checked by the semantic analyser. They include uniqueness rules, identification of identifiers, type consistency rules and special rules. In particular, we have discussed rules from the last group. They are mostly visibility and restriction rules. In each part of a TROLL specification, rules that can be statically checked have been presented. Aspects of the specification that can only be checked dynamically have been mentioned and will be discussed in the next chapters.

Chapter 5

Persistence

Once the syntax and static semantics of a TROLL specification have been analysed, and the specification has been stored in a structured way, the next phase is to generate the appropriate code for animating the specification. Since we want objects involved in an animation session to be persistent, we must create the corresponding data structures which will store the objects' states at run time. This chapter presents, after a short introduction, the persistence model chosen for storing objects created in animation time. The generation of code implementing the behaviour of the objects will be discussed in Chapter 6.

5.1 Introduction

Following the comparison of the development of a TROLL animation environment with the construction of a compiler, the next phase after analysis is the synthesis phase. This phase is concerned with the generation of code from the specification. In the TROLL environment, it means two kinds of activities, as depicted in Fig. 5.1:

- *Database Schema Generation:* A database schema is generated from the specification. The database will contain the object instances created on animation time. The animator will access through an interface to the database for retrieving and updating the state of the objects.

- *Code Generation:* The specification is translated into an executable program. The resulting code is compiled and dynamically loaded into

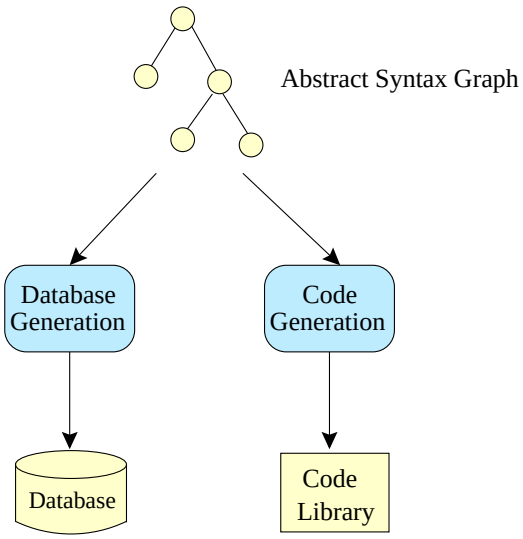


Figure 5.1: Synthesis Phase of TROLL Specifications

the animator. During the animation, an execution manager monitors the execution of state transitions in the system and interacts with the generated code that implements the corresponding actions behaviour.

In this chapter, we describe the transformation of the static structure of TROLL specifications into database schemas. The operational meaning of state transitions in the TROLL model and the generation of code from the specifications will be discussed in the next chapter.

Persistence is a desired requirement when animating large object societies. In the TROLL environment, data representing the states of the objects created in the animator is persistent, i. e. it has a lifespan that is not limited to single executions of the animator. Advantages of having persistence are:

- Only the state of the objects participating in a state transition are loaded and updated. Objects have not to be on main memory all the time during an animation session. Therefore animation sessions can have any duration, and the number of objects involved in a state transition can be large.
- An animation session can be interrupted anytime and with the same objects' states continued later.

- Some scenarios may require the creation of many objects in order to validate user requirements. Once objects have been created, the specification can be validated in several scenarios without creating the objects in each scenario again.

- In a batch execution, persistence allows to observe the final objects' states.

Objects' persistence can be obtained by storing data in system files or by using a database management system (DBMS). The advantages of using a DBMS instead of system files for storing data are evident and have been cited in numerous books [ACPT99, Dat95, Neu96, Ram98]. Here, we enumerate the most important ones for the animation of TROLL models:

- *Data Independence:* The animator is independent from details of data representation.

- *Data Integrity:* The DBMS assures the consistency of the data, for instance, by checking referential integrity constraints among objects.

- *Data Sharing and Concurrent Access:* Various users can animate a specification with the same data. Furthermore, the DBMS guarantees shared access to data by many users operating simultaneously.

- *Efficient Data Access:* The DBMS makes use of a variety of special techniques to store and retrieve data efficiently.

Additionally, since applications specified with TROLL are mostly information systems that require database support, the generated database schemas could be used not only by the animator but also by the applications.

There are different database technologies depending on the model they use for structuring data. Since most current database systems are based on the relational data model, we present in this section how to map the structure of TROLL objects into relational databases (RDBMSs). In fact, Postgres[1], the DBMS used in the TROLL environment, is classified as an object-relational database system (ORDBMS) [Sto94]. Postgres extends the relational data model by some object concepts such as inheritance. A new version of SQL, the standard reference language for relational databases, has

[1]Documentation about Postgres can be found at its homepage located at http://www.postgresql.org

recently been prepared and includes among other new capabilities, object paradigm support [Mel96]. The new version, SQL-3, is still far from being widely adopted. So we do not use any of the new features of SQL-3 in the presentation of the transformation rules. Although the use of a DBMS based on the object-oriented model (OODBMS) [BM93, Loo95] would have entailed a more seamless transformation of the TROLL concepts, it was rejected because of the scarce availability of OODBMSs. Moreover, the object-oriented concepts and functionalities supported by current object-oriented database systems are usually proprietary.

Unfortunately, the representation of object-oriented concepts in the relational data model is not straightforward. In the next, we enumerate which difficulties have to be solved in order to map the static structure of TROLL objects into relational tables:

- *Object Identities:* Objects have an identity, which distinguishes them from all other objects. The identity of an object never changes during its lifetime and is independent of the state, i. e. two distinct objects can have the same state and differ only in the identity. The relational model does not offer such identity concept.

- *Aggregation:* There is no direct support for expressing "has-a" relationships among objects. This has to be represented in a very different form.

- *Specialisation:* The relational model does not provide support for specialisation classes. It is impossible to do a true 1-1 mapping between a relational table and a class when that class inherits attributes from another class.

- *Complex Data Types:* A relation can only have attributes whose types are atomic. Complex data types have to be implemented in different tables related by referential integrity constraints.

- *Object-Valued Types:* References to objects require the localisation of objects in the database as well as referential integrity checking.

Independently of how all these concepts are mapped to relational tables, the database interface provides a persistence layer that encapsulates the access to the database. This means that the animator and the code generated from the specification do not contain any information about the database

design. Since the structure of the database depends on the specification from which the database schemas were generated, the persistence layer cannot directly know which tables must be queried and updated. For this reason, the persistence layer uses a data dictionary for obtaining the information needed to map TROLL objects to tables and generates dynamic SQL based on this information.

We have to take into account two conflicting criteria when designing a database: *performance* and *maintainability*. The harder a data model is optimised for performance, the higher is the maintenance cost in case of changes. High normal forms assure a low maintenance cost but also require the decomposition of the relational schemas. Decomposition implies the use of joins that can be very expensive when retrieving data from several tables in order to build a complex object. On the one hand, to make practical the validation of user's requirements by animation, it is essential that the animator reacts in real time to the user's demands. During an animation, the number of objects involved in a state transition can be relatively large, especially if a multicalling to several objects must be executed. That entails several accesses to the database to load and update the state of each participating object, which can considerably slow down the execution. So performance in the persistence mechanism must be considered. On the other hand, data redundancies should be minimised in order to avoid possible anomalies and to preserve the integrity of the data.

Next, we present how the static structure of TROLL objects is mapped to relational tables.

5.2 Mapping TROLL Objects into the Relational Model

The relational and object-oriented models are conceptually very different. Nevertheless, numerous solutions for mapping object concepts into relational tables have been proposed [Amb99, Kel97, BW95, RB97, Fus97]. That is due to the fact that in most of the current software systems, the programming language is object-oriented and the persistence mechanism is a relational database. The CATC system introduced in Chapter 3 is an example of this[2]. This section describes the mapping rules used by the database schema

[2]The system has been implemented in C++ using Oracle as DBMS.

generator. Details about the implementation in Postgres can be found in [Sch99]. Mapping rules for a previous version of TROLL have been discussed in [Dan95]. Rules are classified according to the concepts they map: global objects, components, specialisations and attributes. Before presenting the rules, we explain how objects are identified in TROLL.

5.2.1 Object Identifiers

Object identifiers provide the unambiguous identification of every object in the system and allow the construction of references between objects. An object identifier is not defined in the class which describes the local behaviour of the object, but in the place in the system hierarchy where it is declared. Identifiers of global objects are declared outside the classes in the object declaration part, whereas identifiers of component objects are defined in the component declaration part of the composite class. A component object has a local identifier inside the composite class and a global identifier formed by the global identifier of the composite object and the local identifier. Consider the next example:

```
object class A
 components Bs(id:nat):B;
 ...
 actions
  actA(p1:nat);
  ...
 behavior
  actA(n)
   do Bs(n).actB, ... od;
end;

object class B
 ...
 actions
  actB;
 ...
end;

object class C
 ...
 actions
  actC(p1:nat,p2:nat);
  ...
end;

objects As(id:nat):A;
objects Cs(id:nat):C;
...
behavior // global behaviour
 Cs(n).actC(p1,p2)
  do As(p1).Bs(p2).actB, ... od;
 ...
end;
```

The specification has three classes `A`,`B` and `C` and two kinds of parametrised global objects `As` and `Cs` of class `A` and `C` respectively. Objects of class `A` have

parametrised components Bs of class B. In the context of class A, component objects are identified by the the local identifier given in the component declaration. So the behaviour specification of actA specifies that whenever the action actA with an input parameter n of an object of class A, for instance As(5).actA(3), is executed, then the action actB of its component Bs(n), i. e. Bs(3).actB, must be executed. On the other hand, component objects have a global identifier which is formed by the composition of its local identifier and the global identifier of the compound. In the example, the global specification part specifies that the execution of the action actC of an object Cs with a parameter identifier n and with input parameters p1 and p2 entails the execution of the action actB in the object with identifier As(p1).Bs(p2). So the execution of As(5).actA(3) and the execution of Cs(n).actC(5,3), where n stands for any parameter value of an existing Cs, require the execution of actB in the same object.
The fact that an object is not identified by its class allows to declare objects of the same class in different places of the system hierarchy. For instance, the specification in the example above could be extended as follows:

objects Bs(id:string):B;

In this case, there are two kinds of objects of class B, those identified by As(n).Bs(m) where n and m stand for natural numbers and those identified by Bs(s) where s represents a string identifier. As a consequence of it, the kind of object identifier referenced by object-valued attributes can only be stated at run time. For example, suppose the class C has an object-valued attribute of class B:

object class C
attributes attB:|B|;
...
end;

The attribute attB is a reference to an object of class B whose identifier type is not statically stated. As we will see later, in order to store references to objects, the database uses synthetic object identifiers (SOIDs) with a uniform simple type for relating tuples of different relations.
Following in a specification the object declarations, and inside each class, the component and specialisation declarations, we can construct and structure the set of possible object instances defined in the specification according

to the place they have in the system hierarchy, i. e. by the way they are identified. Objects in the database are stored following this structure. To make easier the presentation of the mapping rules, we introduce the next definition that classifies the object instances according to the identification structure.

Definition 5.2.1 Let S be a specification, $Id(S)$ represents the set of all possible identification patterns of the object instances defined in S. An identification pattern is given by a list of triples that represent an identification path in the objects hierarchy. The set $Id(S)$ is defined as follows:

1. Let $ObjDecl = (name, ptype, class)$ be an object declaration in S where $name$ is the object name, $ptype$ is the parameter type name or an empty string and $class$ is the class name, then $[(name, ptype, class)] \in Id(S)$.

2. Let $id \in Id(S)$ and let (n, t, c) be the last element in id. Let $CmpDecl = (name, ptype, class)$ be a component declaration of c where $name$ is the component name, $ptype$ is the parameter type name or an empty string and $class$ is the class name then $id \circ [(name, ptype, class)] \in Id(S)$.

3. Let $id \in Id(S)$ and let (n, t, c) be the last element in id. Let $class$ be a specialisation class of c then $id \circ [(\epsilon, \epsilon, class)] \in Id(S)$.

4. Nothing else belongs to $Id(S)$.

□

Id contains all identification paths of the objects that may exist during the execution of the specification. Each element of an identification list includes also the object class name. Actually, the class name is not necessary for identifying a global or component object and can always be found out following the object and component declarations in the specification. We have introduced it, because it makes easier the presentation of the mapping rules. As the parametrisation in object and component declarations is optional, we have introduced the empty string in the parameter type for denoting that the declaration is not parametrised, i. e. it is a declaration of a single object or component. Specialisations are included in the identification path in order to distinguish between properties from the basis class and properties from the specialisation class. In the previous example, Id contains the following identification patterns:

$$Id = \{\ [(\texttt{As,nat,A})],\ [(\texttt{Cs,nat,C})],\ [(\texttt{Bs,string,B})],\ [(\texttt{As,nat,A}),(\texttt{Bs,nat,B})]\ \}$$

Next, we present how the static structure of the objects is mapped into the database.

5.2.2 Global Objects

In the database, objects are mapped into tuples of a relation. A design question is how objects are structured in the database, i. e. in which way they are grouped in tables. In our design, objects are grouped by the place they have in the object hierarchy. That is, there is a table for each identification pattern of the objects in the system. This allows a quick localisation of the objects in the database. Note that it does not necessarily mean to have a unique table for each class in the specification. Since a class may be used in several object declarations, it is possible to have several tables of the same class in the database. Another design approach would consist in creating a unique table for each class in the specification. In this case, additional mechanisms would be necessary for extracting information from a table containing objects with a different identification structure.

Another design issue is how tuples are identified in the tables. A first solution is to use as primary key in a relation the same identification mechanism that is used for the TROLL objects. However, this solution is not suitable for representing references between objects, because it would entail the use of composite keys and tables with a different key structure. A better approach is the use of surrogate keys. A surrogate key is an artificial or synthetic key that is used as a substitute for a natural key. In the object-oriented world, surrogates are usually called synthetic object identifiers (SOIDs). A SOID is a single column attribute, usually a long integer, with no business meaning which uniquely identifies a tuple in a relation. In the assignment of SOIDs there are two levels of uniqueness to be considered: uniqueness within a table and uniqueness across all tables. In order to represent references to objects that may be stored in different tables, we require SOIDs that uniquely identify an object across all tables. SOIDs are managed internally by the DBMS and are not visible in the animator. The DBMS assigns a SOID to an object in the moment of its creation, i. e. when a tuple is inserted in the corresponding relation. There are several techniques for implementing the generation of SOIDs. Most current DBMSs have built-in features for generating unique

values that can be used for SOIDs values. A SOID within a table can also be calculated by using the SQL MAX() function on the SOID column and then adding one to it. Other techniques include using an additional table with counters as columns. We do not go here into more details. For more information about key generation we refer the reader to [Amb99, Dan95].

Another matter is the selection of table names. During the animation, the database interface makes use of the table names for identifying objects in the database. So during the database schema generation, the object's name given in a global object declaration is assigned to the table which will contain the instances identified by the same name. Furthermore, since in a specification the names of global objects must be distinct from each other, the uniqueness of the table names is assured. However, the assignment of names to tables representing component objects presents the problem that the uniqueness of a component name is only guaranteed in the class where the component is declared. A solution is to use as name the composition of all the names included in the identification path. A problem with this solution is that table names can become relatively long and therefore exceed the length allowed by the DBMS. A better solution is to use the component name concatenated with a number generated by a counter. Whenever a component name has been already used, the counter is incremented. The table hierarchy is then stored in an additional table. At animation time, the database interface uses this table for finding out in which table a component for a given object is stored.

The next rule defines how global objects are mapped into the database. It makes use of the identification structure defined in Def. 5.2.1. In the definition of rules, we adopt the following conventions: $\mathcal{R}_{\text{TROLL}}(S)$ denotes the set of relation schemas generated for a specification S; the attributes that make up the primary key in a relation are underlined; the uniqueness constraint on the value of a set of attributes $att_1, \ldots, att_n$ is represented by `unique(`$att_1, \ldots, att_n$`)`, and the *not null* constraint on an attribute att is denoted by `not_null(`att`)`. We assume the domain of SOIDs as already defined and denoted by `soid_dom`.

Rule 5.2.1 Given a specification S and an identification path in S, $id \in Id(S)$, such that $id = [(name, ptype, class)]$, then exists a relation $R_{name} \in \mathcal{R}_{\text{TROLL}}(S)$ defined as follows:

if $ptype = \epsilon$ then

$$R_{name} = (\underline{\texttt{soid}}\texttt{:soid\_dom})$$

otherwise

$$R_{name} = (\underline{\texttt{soid}}\texttt{:soid\_dom}, \texttt{par:}ptype)$$

and (`unique(par)` $\wedge$ `not_null(par)`)

□

Objects declared as *global* are those with only one element in the identification path. For each global object declared, there is a relation in the database schema. Parametrised objects have the parameter identifier as attribute in the corresponding relation. As parametrised objects must always have a parameter identifier and it must be unique, the corresponding attribute in the relation is declared as `not_null` and `unique`. The extension of the relations with attributes of the classes will be described later. Next, we present the transformation of component objects.

5.2.3 Components

As global objects, component objects are structured into tables. In the database, the relationship between component and composite objects must be represented by relationships between the rows of the respective tables. This can be done through referential integrity constraints, also called foreign keys constraints. A foreign key is a set of data attributes that appears in one table and must hold a key to another table. In TROLL, a component relationship can be seen as one-to-one or one-to-many relationship, depending if the component was declared as single or parametrised, whereby the component object cannot be shared and depends existentially on the compound object. The usual way of mapping such relationship is by including the key of the composite table as a foreign key in the component table [Amb99]. In this way, it is guaranteed that the existence of the component depends on the existence of the composite. The mapping is defined in Rule 5.2.2. In order to avoid conflicts in the relation names, we assume that for a given component identifier "name", there is a renaming "name$_r$" which uniquely identify the component. As usual, referential integrity is indicated by an arrow "$\rightarrow$".

Rule 5.2.2 Given a specification S and an identification path for a component in S, $id \in Id(S)$, such that $id = [e_1, \ldots, e_n]$ and $e_n = (name, ptype, class)$ and let $R_{compl} \in \mathcal{R}_{\text{TROLL}}(S)$ be the relation associated to the path of the composite, $(id_{compl} = [e_1, \ldots, e_{n-1}])$, then exists a relation $R_{name_r} \in \mathcal{R}_{\text{TROLL}}(S)$ defined as follows:

if $ptype = \epsilon$ then

$$R_{name_r} = (\underline{\texttt{soid}}\texttt{:soid\_dom}, \texttt{complex:soid\_dom})$$

and ($\texttt{complex} \rightarrow R_{compl}(\texttt{soid}) \wedge \texttt{not\_null(complex)} \wedge$ `unique(complex)`)

otherwise

$$R_{name_r} = (\underline{\texttt{soid}}\texttt{:soid\_dom}, \texttt{complex:soid\_dom}, \texttt{par:}ptype)$$

and ($\texttt{complex} \rightarrow R_{compl}(\texttt{soid}) \wedge \texttt{unique(complex,par)} \wedge$ `not_null(complex)` $\wedge$ `not_null(par)`)

□

The attribute `complex` is a foreign key to the primary key attribute of the composite relation. Since a component object cannot exist autonomously, the attribute `complex` is defined as `not_null`. In single components, a `unique` constraint on `complex` expresses that a composite can only have a component. In case of multiple components, the parameter identifier is included in the relation and defined as `not_null`. The uniqueness of a parameter identifier is expressed by the `unique` constraint on the attributes `complex` and the parameter identifier `par`.

Note that by multiple components, the parameter identifier is included in the component table although in TROLL, it is defined in the composite. The natural solution would require an additional relation representing the component relationship. The new relation would have as primary key the soid of the complex object and the parameter identifier of the component. Additionally, the relation would have the soid of the component as attribute:

$$R_{rel} = (\underline{\texttt{soid\_complex}}, \underline{\texttt{par}}, \texttt{soid\_compon})$$
$$R_{compon} = (\underline{\texttt{soid}})$$

Since a component object must always belong to a composite object and only one, the attribute in R_{rel} representing the soid of the component is a candidate key in the relation. For this reason, we have merged R_{rel} in the component relation with no risk of redundance because there is a one-to-one correspondence between the respective instances.

In case of single components, the embedding of the soid could also be in the other direction, i. e. the composite relation would have an attribute with the soid of the component. Nevertheless, this solution would mean a non-uniform database design because single and multiple components would be then accessed in different ways.

As already mentioned, there is a special table which includes the hierarchy structure of the tables. This table is used by the persistence layer for finding out in which table the data for a giving identification path is stored. We do not present its construction here but assume the schemas generator creates and instatiates it at the same time the other tables are constructed.

The next example illustrates the mapping of component objects following the rule 5.2.2.

Example 5.2.1 Given the following specification:

```
object class A
 components Bs(id:nat):B;
 ...
end;
object class B
 components Cs(id:nat):C;
 ...
end;
```

```
object class C
 ...
end;
objects As(id:nat):A;
```

The resulting database schema contains the following relations:

$$R_{As} = (\underline{\texttt{soid}},\texttt{par})$$
$$R_{Bs_1} = (\underline{\texttt{soid}},\texttt{complex} \rightarrow R_{As}(\texttt{soid}),\texttt{par})$$
$$R_{Cs_1} = (\underline{\texttt{soid}},\texttt{complex} \rightarrow R_{Bs_1}(\texttt{soid}),\texttt{par})$$

□

In the animator, component objects are not retrieved together with the compound object, but only when they are necessary. Whenever an object is loaded into main memory, the object gives its global identifier to the persistence layer in order to retrieve the data. The persistence layer then must find the corresponding row in the corresponding table in which the values of the attributes are stored. To this end, it is necessary to access all tables involved in the identification path, either through joins or through nested queries. In the example above, the access to the attributes of an object, for

instance, `As(2).Bs(7).Cs(3)`, would require the access to the three tables. For long identification paths, the access to objects' attributes could result in a bottleneck. A more efficient solution, although poorer in design, would consist in embedding in the component table not the soid of the composite but all parameter identifiers of the identification path. In this case, the global idenfication of an object can be built with the information included in the row and with the help of the hierarchy table. The solution is optimal in terms of performance as the tables of the composites do not need to be accessed. On the other hand, that would result in data redundancies. Next, we define the rule associated to this solution.

Rule 5.2.3 *(alternative to Rule 5.2.2)* Given a specification S and an identification path for a component in S, $id \in Id(S)$, such that $id = [e_1, \ldots, e_n]$ and $e_n = (name, ptype, class)$ and let $R_{compl} \in \mathcal{R}_{\text{TROLL}}(S)$ be the relation associated to the path of the composite, $(id_{compl} = [e_1, \ldots, e_{n-1}])$. Let $Par(id_{compl}) = [ptype_1, \ldots, ptype_m]$ be the list of parameter types in id_{compl} different from the empty string, then exists a relation $R_{name_r} \in \mathcal{R}_{\text{TROLL}}(S)$ defined as follows:

if $ptype = \epsilon$ then

$$R_{name_r} = (\underline{\texttt{soid}}\texttt{:soid\_dom}, \texttt{par}_1\texttt{:}ptype_1, \ldots, \texttt{par}_m\texttt{:}ptype_m)$$

and $((\texttt{par}_1, \ldots, \texttt{par}_m) \rightarrow R_{compl}(\texttt{par}_1, \ldots, \texttt{par}_m) \wedge$
$\texttt{unique(par}_1\texttt{,} \ldots \texttt{,par}_m\texttt{)} \wedge$
for $1 \leq i \leq m$: $\texttt{not\_null(par}_i\texttt{)})$

otherwise

$$R_{name_r} = (\underline{\texttt{soid}}\texttt{:soid\_dom}, \texttt{par}_1\texttt{:}ptype_1, \ldots, \texttt{par}_m\texttt{:}ptype_m, \texttt{par}_{m+1}\texttt{:}ptype)$$

and $((\texttt{par}_1, \ldots, \texttt{par}_m) \rightarrow R_{compl}(\texttt{par}_1, \ldots, \texttt{par}_m) \wedge$
$\texttt{unique(par}_1\texttt{,} \ldots \texttt{,par}_m\texttt{,par}_{m+1}\texttt{)} \wedge$
for $1 \leq i \leq m+1$: $\texttt{not\_null(par}_i\texttt{)})$

□

As usual, the rule distinguishes between single and parametrised components. A foreign key to the parameter identifiers included in the composite assures the existencial dependence of the component. The uniqueness of the parameter identifiers is expressed by the `unique` constraint. All parameter identifiers are declared as `not_null`. Note that with this rule, the existential dependence can only be expressed through foreign keys if the composite relation has at least a parameter identifier attribute. If this is not the case, the referential integrity has to be guaranteed by either database triggers or by the persistence layer. Next example illustrates the mapping rule.

Example 5.2.2 Given the specification of example 5.2.1 and using the mapping rule 5.2.3, the resulting database schema contains the following relations:

$$R_{As} = (\underline{\texttt{soid}},\texttt{par}_1)$$
$$R_{Bs_1} = (\underline{\texttt{soid}},\texttt{par}_1 \rightarrow R_{As}(\texttt{par}_1),\texttt{par}_2)$$
$$R_{Cs_1} = (\underline{\texttt{soid}},(\texttt{par}_1,\texttt{par}_2) \rightarrow R_{Bs_1}(\texttt{par}_1,\texttt{par}_2),\texttt{par}_3)$$

□

5.2.4 Specialisations

Class specialisations are also mapped into tables. Each table includes the attributes specific to the subclass. As specialised objects are also objects of the superclass, the rows in the basis and specialisation tables containing information about the properties of the same object are identified with the same soid. That is, the primary key of the table that represents the subclass is a foreign key to the primary key of the basis table. The mapping rule is defined as follows.

Rule 5.2.4 Given a specification S and an identification path for a specialisation in S, $id \in Id(S)$, such that $id = [e_1, \ldots, e_n]$ and $e_n = (\epsilon, \epsilon, class)$ and let $R_{sup} \in \mathcal{R}_{\text{TROLL}}(S)$ be the relation associated to the upper path, $id_{sup} = [e_1, \ldots, e_{n-1}]$, then exists a relation $R_{class_r} \in \mathcal{R}_{\text{TROLL}}(S)$ defined as follows:

$$R_{class_r} = (\underline{\texttt{soid}}\texttt{:soid\_dom}) \text{ and } \texttt{soid} \rightarrow R_{sup}(\texttt{soid})$$

□

The name of the relation is given by the corresponding renaming of the class name. The foreign key relates tuples from the basis and specialisation tables that refer to the same object. The class inheritance tree is also stored in the hierarchy table. A disadvantage of this mapping approach is that retrieving an object with specialisations requires several database operations because its attributes are contained in different tables.

Another technique for mapping specialisations in a relational database consists in including in the basis table all attributes of the specialisation classes. So no additional tables are necessary for the subclasses. The attributes of unused specialisations are filled with NULL values. The advantage of this approach is that it provides a good performance because all of the data about an object is found in one table and can therefore be accessed with a single database operation. The drawback is that it could lead to a substantial number of NULL columns in the table, wasting space. Moreover, it has to be found out in which aspects an object is specialised. This information could be contained in an additional column in the table.

There is another technique which also uses a different table for each specialisation class. In this case, each table includes both the attributes and the inherited attributes of the class that it represents. This approach provides fast performance because all the data about an object is stored in only one table. However, there are several drawbacks. Queries about generic properties of all objects in the hierarchy tree require the access to all the tables. Furthermore, the modification of a class results in a high maintenance cost because not only its table but also the tables of its subclasses must be modified. Yet another problem with this technique is that if an object has several specialisation aspects, then the attributes inherited from the superclass are contained in each of the specialisation tables. So additional mechanisms are required for maintaining the data integrity.

For a more detailed discussion of these techniques the reader is referred to [Amb99, Kel97, RBP+91].

5.2.5 Attributes

Once the relations for storing the objects and their relationships have been defined, we can extend them with the attributes that represent the state of the objects. The way attributes are mapped into the database depends on the data type they have. Since the relational model requires that all attributes in a relation have elementary domains, multivalued attributes must be mapped

into separate relations. In the presentation of the mapping rules, we do the distinction between attributes that can be directly introduced as attributes in the relation and those that need a different relation. From the former group are those with the following data type:

- *Standard Elementary Data Types:* TROLL standard atomic data types are directly represented in the relational model. The specific implementation of these types depends on the DBMS. If a type is not provided directly by the DBMS, it can usually be defined by predefined types with additional domain constraints. For instance, natural numbers can be defined as positive integers plus the zero.

- *Enumerations:* An enumeration type can be represented by the string type plus an integrity constraint that limits its values to the label names defined in the enumeration. Actually, labels could be represented as numbers, as they are usually represented in programming languages. A problem with this is that in the animator, enumeration values must be presented to the users by the label names. In this case, the mapping from the numbers into the corresponding label names should be done additionally.

- *Object-References:* Object-valued attributes are represented in the database by attributes containing the SOIDs of the referenced objects. Since objects of the same class may be stored in different relations, it is not possible to ensure the referential integrity by directly using a foreign key to the referenced object. A foreign key is constrained to *point* to an object in a particular referenced relation. A possible solution is the use of an intermediate relation. For each class, an additional relation is created that contains the SOIDs of the existing objects of the class. An additional attribute in the relation indicates the relation where the object is located. An object-valued attribute is then a foreign key to the primary key of the relation containing the SOIDs of the respective class. As we will see later in Sect. 7.2.6, the database generator uses triggers for implementing referential integrity constraints. The use of triggers does not require any additional table including the SOIDs of objects of the same class. Since triggers are not supported by the SQL-92 standard, we do not use them in the transformation rules.

- *Records:* The record constructor allows the definition of types whose instances are tuples of values of possibly different types. In the database,

each record field is mapped into an attribute of the relation.

As integrity constraints, attribute features are not mapped into the database. They are transformed, if necessary, into C++ code by the code generator and checked at animation time. The fact that a hidden attribute of an object cannot be accessed by other objects was already statically checked by the semantic analyser. The constraint that a constant attribute may only be assigned at the moment of the creation of the object was also checked during the analysis phase. The constant feature of an attribute could also be assured by the DBMS through transitional constraints, in case they are supported. Assignments to initialised and derived attributes are directly handled by the animator. Since the derivation terms of derived attributes can be arbitrarily complex, derived attributes are also stored in the database. The value of a derived attribute is only computed when the values of the attributes, on which its value depends, may have been changed. The only attribute constraint mapped into the database is that non-optional attributes must always have a value. This is represented by a `not_null` constraint on the attributes of the relation.

Next, we present the mapping rule for the attributes cited above. For simplifying the definition of the rules, we assume that the class of a relation is the class given in the last element of the identification path from which the relation is generated.

Rule 5.2.5 Given a specification S and let $R \in \mathcal{R}_{\text{TROLL}}(S)$ be a relation of class C. Let $attrDecl = (name, type)$ be an attribute declaration in C where $name$ is the attribute name and $type$ the attribute type, then R is extended as follows:

- if $type$ is a standard elementary type then

$$R = (\underline{\texttt{soid}}, \ldots, name : type)$$

- if $type$ is an enumeration type with labels $(label_1, \ldots, label_n)$ then

$$R = (\underline{\texttt{soid}}, \ldots, name : \texttt{string})$$

  and $\forall r \in R : (r.name \in \{\text{“}label_1\text{”}, \ldots, \text{“}label_n\text{”}\})$

- if $type$ is an object-valued type of class C_1 then

$$R = (\underline{\texttt{soid}}, \ldots, name : \texttt{soid\_dom})$$

and $name \rightarrow R_{C_1\_Ids}(\texttt{soid})$ where $R_{C_1\_Ids}$ is the relation containing the soids of the instances of class C_1

- if *type* is a record type with fields $r_1 : type_1, \ldots, r_n : type_n$ then

$$R = (\underline{\texttt{soid}}, \ldots, name_{r_1} : type_1, \ldots, name_{r_n} : type_n)$$

for $1 \leq i \leq n$, if $type_i$ is a enumeration, an object-valued or a collection type, then apply the corresponding mapping rule.

If the attribute is not an optional attribute then `not_null(`*name*`)`

□

Attributes containing collections of values are mapped into different tables. As specialisations, relations between tables storing data of the same object are maintained by foreign keys. The primary key of the table representing the complex attribute is a foreign key to the primary key of the table in which the object is stored. As the hierarchy table, there is an additional table that stores the table names where the complex attributes are located. During the animation, the persistence layer accesses to this table in order to know the tables in which the complex attributes of a given object are stored.

In TROLL, constructors that allow the definition of types whose instances are collection of values (of the same type) are:

- *Sets:* A set is a non-ordered collection of values without duplicates. Since sets cannot have duplicates, the attribute in the relation that stores the values is included in the primary key.

- *Bags:* A bag is also a non-ordered collection of values, but allowing duplicates. Bags are mapped like sets with an additional attribute that contains the number of occurrences of each value in the bag.

- *Lists:* A list is an ordered collection of values, possibly with duplicates. In the corresponding relation, the position of each element in the list is stored in an attribute. This attribute is included in the primary key.

- *Maps:* A map is a non-ordered collection of pairs of values, *(key,value)*, without duplicated keys. Maps can be seen as sets with an additional value associated to each element. So maps are represented like sets with an additional attribute containing the value associated to each key.

Next, we define the mapping of collections in rule 5.2.6. Note that collections can be defined recursively, resulting in data structures of arbitrary complexity. Because of this complexity we do the restriction that in the animation environment, collections can only have atomic or record values. Actually, the complexity lies in the access to the data structure and its visualisation in the user interface of the animator.

Rule 5.2.6 Given a specification S and let $R \in \mathcal{R}_{\text{TROLL}}(S)$ be a relation of class C. Let $attrDecl = (name, type)$ be an attribute declaration in C, where *name* is the attribute name and *type* is the attribute type. If *type* is a collection type, then exists a relation $R_{name_r} \in \mathcal{R}_{\text{TROLL}}(S)$ defined as follows:

- if *type* is a set of type $type_1$ then

$$R_{name_r} = (\underline{\texttt{soid}} : \texttt{soid\_dom}, \underline{name} : type_1)$$

  and $\texttt{soid} \rightarrow R(\texttt{soid})$

- if *type* is a bag of type $type_1$ then

$$R_{name_r} = (\underline{\texttt{soid}} : \texttt{soid\_dom}, \underline{name} : type_1, \texttt{num} : \texttt{nat})$$

  and $(\texttt{soid} \rightarrow R(\texttt{soid}) \wedge \texttt{not\_null(num)})$

- if *type* is a list of type $type_1$ then

$$R_{name_r} = (\underline{\texttt{soid}} : \texttt{soid\_dom}, \underline{\texttt{pos}} : \texttt{nat}, name : type_1)$$

  and $(\texttt{soid} \rightarrow R(\texttt{soid}) \wedge \texttt{not\_null}(name))$

- if *type* is a map of type $(type_1, type_2)$ then

$$R_{name_r} = (\underline{\texttt{soid}} : \texttt{soid\_dom}, \underline{name_{key}} : type_1, name_{val} : type_2)$$

and ($\texttt{soid} \rightarrow R(\texttt{soid}) \wedge \texttt{not\_null}(name_{val})$)

If the type of the elements in the collection is an enumeration, an object reference or a record type, then apply the mapping according to rule 5.2.5. □

Note that non-optional multivalued attributes do not require any additional constraint assuring that the relation has at least one tuple because collections may have the value "empty". The restriction that a collection cannot be empty can be specified as an integrity constraint in the class.

5.3 Summary

In the TROLL animator, objects are persistent, i. e. they have a lifespan that is not limited to single executions of the animator. Animation sessions can then be interrupted and with the same object population continued later. Objects are stored in a relational database. The representation of object-oriented concepts in the relational data model is not straightforward. In this chapter, we have analysed how the static structure of TROLL specifications can be transformed into relational database schemas. A set of mapping rules have been presented. During the synthesis phase of the TROLL animator, the database schemas are generated from the specifications according to these rules. The mapping rules are summarised as follows:

- Objects are grouped into tables by the place they have in the object hierarchy. That is, there is a relation for each object and component declaration in the specification. This allows a quick localisation of the objects in the database.

- Surrogate keys (SOIDs) are used as primary keys in the relations. By this way, all tuples in the database are identified uniformly and with a single column attribute. This allows a simple and uniform mechanism for relating tuples of different relations. Surrogate keys are managed internally by the persistence layer and are not visible in the animator.

- The relation between a component and its composite is represented by including a foreign key in the component table that refers to the primary key of the composite table.

- Specialisation classes are mapped into tables. Each table includes only the attributes specific to the subclass. The relation between the tables representing the base and specialisation properties of the objects is represented by using as primary key in the specialisation table a foreign key to the primary key of the base table. That is, rows in the base and specialisation tables containing information about the same object have the same identification value.

- Attributes with simple data types declared in the TROLL classes are directly introduced as attributes in the corresponding relations. An object-valued attribute contains the SOID of the referenced object and is a foreign key to the primary key of a relation containing the SOIDs of the referenced class. For attributes of type record, each record field is mapped into an attribute of the relation. Attributes containing collections of values are mapped into different tables. Similar to specialisations, the relation between rows of the tables storing data of the same object is maintained by foreign keys.

- Additional tables contain information about the tables hierarchy. During the animation, the persistence layer uses these tables to localise objects in the database.

The access to the database is encapsulated by the persistence layer. So the animator does not know about how objects are stored into the database. In order to retrieve/update data of an object, the animator just gives the global identifier of the object and the required operation to the persistence layer. The database represents the static structure of the specifications. The implementation of the actions behaviour, instantiation of initialised attributes, valuation of derived attributes and the check of integrity constraints defined in the classes will be presented in the following chapter.

Chapter 6

Behaviour

After the generation of the database schema, the next step in the synthesis phase is the generation of code that implements the behaviour of the objects. In order to generate the appropriate code, we have first to analyse the operational meaning of state transitions in TROLL and determine an execution model. The code must then be generated according to the execution model. This chapter is concerned with the behaviour of TROLL specifications and its execution. In Sect. 6.1, the dynamics of TROLL objects are analysed and an execution model is given. The implementation of the execution model and the translation of the specification into an object oriented programming language are described in Sect. 6.2.

6.1 Execution Model

An object system is a community of independent communicating objects. The life cycle of an object consists of a sequence of events. A possible behaviour of a system is given by the life cycles of the objects belonging to the system. These life cycles are independent from each other as long as the corresponding objects do not communicate. In case of communication, objects share an event in their life cycles to perform synchronously actions. Events are occurrences of actions that change the state of the objects. The state of an object is given by the current value of its attributes. In the animator, users simulate an event by selecting the execution of an initial action in a determined object establishing a state transition in the system. The execution of the initial action may imply the execution of other actions through

action calling. Action calling is defined as a transitive, asymmetric, synchronous relation. Action calling bases on a synchronous communication model easy to formalise on a logic level. The calling relation may relate actions of the same object or actions of different objects. The latter denotes a shared event in the life cycles of the objects. Aspects to be taken into account when executing the action chain given by a calling relation are:

- *Parallel Execution:* Conceptually, all actions involved in the calling relation are atomic in duration. That is, all operations to be executed occur in one instance of time.
- *No Conflict in Attribute and Variable Assignments:* Values assigned to attributes and variables must be consistent. This means that different values cannot be assigned to the same attributes or variables.
- *Atomicity:* All actions happen in an indivisible unit of execution. If an action cannot take place, for instance, because its precondition is not satisfied, then no actions of the action chain can take place.
- *Consistency:* Consistency demands that the carrying out of the actions does not violate any of the integrity constraints defined on the objects.
- *Termination:* The set of actions to be executed must be finite.
- *Isolation:* All actions involved in a calling relation must be executed in isolation. That is, the execution of several initial actions must not interfere.

As can be observed, the execution of an action chain is similar to the execution of ACID transactions in databases. In what follows, we discuss the execution of the calling relation given by an initial action. Action calling for a previous version of TROLL has been discussed in [HS93, Har95, Ner95]. We assume the isolation property of the calling relation. The reason for this lies on the fact that during the animation, the objects' states are stored in a database. Thus the isolation property of the calling relation is guaranteed by the concurrency control system of the DBMS.

6.1.1 Parallel Execution

At a conceptual level, the execution of all statements (assignments and action callings) defining the behaviour of an action occur in parallel. Seman-

tically, there is no problem with this requirement because, as indicated in Sect. 3.4, the behaviour of an action is given by the conjunction of a set of logic formulas that must be satisfied whenever the action occurs, and where the computation order of the formulas is irrelevant. However, if we want to execute the specification using a sequential programming language, in our case C++, we have to choose an execution order of the statements defined in the behaviour of actions. Since a statement may use local variables that can be instantiated by other statements, we must take into account such data flow when selecting the execution order. Next example illustrates this case.

Example 6.1.1 Given the following TROLL specification:

```
object class A
  ...
  actions
    act1(n:int);
    act2(n:int,m:int);
    act3(!n:int); /* '!' denotes an output parameter */
  behavior
    act1(n)
      var a:int;
      do act2(n,a), act3(a), ..., od
    ...
end;
objects objA : A;
```

Note that unlike imperative programming languages, the syntactic sequence of the statements does not determine the sequence in which they must be computed. So if we want to execute the action `objA.act1`, a correct execution order would be `act3, act2`, but not `act2, act3` because `act2` has an input parameter, `a`, whose value is determined by `act3`. □

A correct execution order must guarantee that a statement can only be executed if all variables that appear in the statement either have a value or are instantiated by the statement. For this, dependencies between statements must be analysed and an execution order assuring the appropriate data flow must be determined. In the TROLL environment, this can be done in the syntax graph containing the specification. Once an execution order has been found, the place of the nodes in the graph are interchanged accordingly. In

this way, during the translation of the specification into the respective program code, the code generator can assume that the positions of the nodes in the graph represent a valid execution order. To find a correct execution order, we can construct a directed graph of dependencies for each action behaviour. Vertexes of this graph are the different statements. Edges between nodes represent data dependencies between statements. Since conditional statements may be nested and may also include several substatements, they have to be unfolded in simple conditional statements before constructing the dependency graph. The following algorithm performs a topological sort of the vertexes and determines the sequence of execution. Additionally, the algorithm checks that all variables may be instantiated during the execution:

Algorithm 6.1.1 Let G be a dependency graph for an action A. Let V be the set of variables that are or may be instantiated during the execution of A. A valid execution order for the statements of A is given by the list L which is constructed as follows:

1. Check for cycles in G. If any then `exit("found mutual dependency")`.

2. While there are vertices in G do

   (a) Choose a vertex v in the graph without output edges. Check if all variables that appear in the corresponding statement either are included in V or are/may be instantiated in the statement. If this is not the case then `exit("found variable not instantiated")`.

   (b) Add the statement of the vertex v at the end of L. Introduce in V the variables that are or can be set by the statement. Delete the vertex v and its input edges from G.

□

Data flow dependencies establish a partial order between the statements. So it is possible that for an action there exist several correct execution orders. This is expressed in the algorithm when the graph has several vertexes without output edges and a vertex has to be chosen. The selection of an execution order from all possible valid execution orders does not entail any indeterminism. Since in a state transition, only a unique value may be assigned to an attribute or a variable, any valid execution order leads to the same state.

Note that V represents the set of variables that are or *may be* instantiated during the execution. The reason for that lies on the fact that the execution of statements can be conditioned by formulas. Since the satisfaction of formulas can be state-dependent, it is not possible to statically know whether a variable will be instantiated or not. So run-time checks are necessary.

As mutual dependencies between statements are possible, the existence of cycles in the graph must be checked. This can, however, restrict the execution of valid TROLL specifications. Consider the next example.

Example 6.1.2 Given the following TROLL specification:

```
object class A
  ...
  actions
    act1;
    act2(n:int,!m:int);
    act3(n:int,!m:int);
  behavior
    act1
      var a:int, b:int;
      do act2(a,b),
         act3(b,a), ...
      od;
    act2(n,m)
      do m := 5, ...od;
    act3(n,m)
      do m := 3, ...od;
    ...
end;
objects objA : A;
```

In this specification, it is not possible to find an execution order for the action `objA.act1` because the data flow goes from `act2` to `act3` and vice versa. Nevertheless, this example is a correct specification in TROLL. Conceptually, this specification means that `act2` uses a variable that is set by `act3` and sets a variable that is used by `act3`. Of course, the specification would not be correct if the values of the variables depended on each other. □

A solution to the problem of finding an execution order for statements with mutual dependencies consists in ignoring the control flow and constructing a global dependency graph with all statements included in the actions that are called. That is, action calling statements would be replaced by the statements of the actions to be called. The same algorithm would also be used for finding a valid execution order. In the example above, the assignments to the output variables of `act2` and `act3` would then be executed before the variables are used. The problem with this solution is that it violates the essence of action calling and object encapsulation. Furthermore, we have to take into account

that mutual dependencies may happen not only between actions of the same object, but also between actions of different objects that can be distributed in several machines.

Another solution to this problem consists in automatically breaking the mutual dependency by converting one of the actions in two actions: one that delivers the required values and another one that uses the variables set by the action causing the mutual dependency. For instance, in the example above, `act2` could be rewritten in the following actions: `act2_1(!m:int)` and `act2_2(n:int)`, where the behaviour of `act2_1(m)` would just contain the assignment to the output variable (`m := 5`) and the behaviour of `act2_2(n)` would contain the rest of statements of the original action. A correct execution order for `act1` would be then `act2_1(b), act3(b,a), act2_2(a)`. A similar approach based on the automatic insertion of *pre-actions* is adressed in [HS93]. There are several drawbacks with this technique. First, finding the correct conversion can be a very complex task. Note that mutual dependencies are not restricted to only two action callings but can involve several ones. Moreover, the values of output variables must not necessarily be assigned directly but they can also be set by other actions trough action calling statements. Second, it should be proved that the new specification corresponds to the original one. For all these reasons, we think the best solution is just to warn users that an execution order could not be found and let them to change the specification in case they want to use the animator.

An issue to take into account is that, as already mentioned, the behaviour of an action can be extended in composite and specialisation classes and also in the global behaviour specification part. If we want to maintain the object encapsulation in the generated code, behaviour extensions must also be translated using a similar structure. So an execution order must also be determined between the local behaviour of an action and the different behaviour extensions. Note that this case and the calling relation are not the same because there is not any explicit calling. Since there is no control or data flows between the local behaviour of an action and its extensions, any execution order is valid. We will just choose an execution order when transforming the specification into C++.

6.1.2 Conflicts in Attribute and Variable Assignments

Atribute values may be changed by the occurrence of actions. In a state transition, changes on attributes values must be consistent. In a sequential

execution of state changes, as in our case, inconsistencies in the attributes assignments would lead to an indeterminism in the new state where the attribute values depend on the execution order that was chosen. So conflicts in attribute assignments must be checked. A syntax-based analysis of these conflicts would be desirable, but since the execution of a statement may be conditioned, this analysis is possible only in some cases. Furthermore, the satisfaction of the condition formulas may be state-dependent. In [HS93] a syntax-based analysis is proposed consisting in ignoring the conditions and then checking possible conflicts in the attribute assignments. The result of this checking would guarantee consistent state transitions independently of the state. However, it could be more restrictive than necessary. Consider the following example:

Example 6.1.3 Given the following TROLL specification:

```
object class A
  attributes
    a:int;
    b:int;
  actions
    act1(n:int);
    act2(n:int);
  behavior
    act1(n)
    do
     b:=40,
     if a < n then act2(n) fi
    od;
    act2(n)
    do
     if a >= n then b:=60 fi
    od;
  ...
end;
objects objA : A;
```

Independently of the current state, the execution of `objA.act1` does not present any conflict in the assignment to the attribute `B`. As can be observed, `act2` only changes the value of `B` if `a >= n`, but `act2` is only called by `act1` if `a < n`. A syntax check as depicted above would however reject the specification. Note that if we change the conditional of `act1` to `a<=n` then there would be an inconsistency that is state-dependent. The inconsistency would occur only if `a` is equal to `n`. □

Therefore run time checks are necessary in order to assure that there are no conflicts in the attribute assignments. These checks must also be extended for local and output variables. Thus during the execution, whenever an attribute

or variable is assigned, it must be checked that the attribute or variable was not already set to a different value by other statements.

6.1.3 Termination

The execution model of action calling is set-based. That is, independently of the number of times an action instance[1] appears in an action chain, it occurs at most once. The calling relation just establishes a transitive implication between occurrences of actions in a state transition. This contrasts with the typical procedure calling of imperative programming languages. Consider the next example.

Example 6.1.4 Given the following TROLL specification:

```
object class A
  ...
  actions
    act1; act2; ...
  behavior
    act1 do act2, ...od;
    act2 do act1, ...od;
  ...
end;
objects objA : A;
```

In this specification, the actions behaviour specifies that whenever `objA.act1` occurs then `objA.act2` must also occurs and vice versa. Note that the direct translation of this specification into an imperative programming language would result in an infinite procedure calling. □

Since an action instance may be executed only once during the execution of an action chain, the names and parameter values of the actions that are executed up to that moment must be stored. Each time an action is called, it must be checked whether the action was already executed.

Although the execution model of action calling is set-based, the termination of a state transition is not guaranteed. Since actions may have input parameters, it is possible that the actions set contains infinite action instances.

[1] Two action instances are the same if they occur in the same object and have the same parameter values.

This is the case, for instance, when a direct or transitive recursive action calling is defined and the end condition is not correct. The next example illustrates this case.

Example 6.1.5 Given the following TROLL specification:

```
object class A
  ...
  actions
    act1(n:nat);
  behavior
    act1(n)
      do
        if n < 5 then ...
        else act1(n+1) fi
      od;
    ...
end;
objects objA : A;
```

As can be observed, the execution of `objA.act1` with a parameter value greater than or equal to `5` results in an infinite action chain and therefore in an endless state transition. □

Another case where the finiteness of the action chain is not guaranteed happens when birth actions call (directly other transitively) other birth actions. Since objects and components may be parametrised over an infinite set of values such as the natural numbers, it is possible that a state transition entails the creation of an infinite number of objects. See the following example.

Example 6.1.6 Given the following TROLL specification:

```
object class A
  ...
  actions
    *create;
    ...
end;
objects As(n:nat) : A;
behavior
```

```
  As(n).create
    do
      if n < 5 then ...
      else As(n+1).create fi
    od;
end;
```

In this case, the creation of a parametrised object `As` with a parameter value greater than or equal to `5` creates in turn an infinite number of objects. Note that the birth action `create` is always called in a different object. □

Again, the presence of state-dependent conditions does not allow a syntax-based check of the finiteness of an action chain. Furthermore, finiteness of action chains is undecidable. In order to assure the termination of state transitions, the cardinality of the actions set mut be restricted:

- For each object instance and for each action, only a finite number of instantiations is allowed.
- For each object class only a finite number of objects may be created.

Checking these conditions requires the handling of several action counters. As it is possible that correct TROLL specifications are rejected depending on the upper boundaries, users and not the animator should fix them. A more restrictive solution, but easier to implement, consists in restricting the number of action instantiations just to one. In this case, it would not be necessary to store the parameter values for each executed action in the action chain. Nevertheless, we find this solution very restrictive. In the next section, we show how the finiteness condition of action chains is implemented in the animator.

6.1.4 Atomicity

The execution model of action calling follows an *all or nothing* approach. That is, either all actions of the calling chain can be carried out or no actions can take place. There are several situations where an action cannot be carried out:

- A precondition does not hold.
- The termination condition is violated.

- There are conflicts in attribute or variable assignments.
- Run time errors are detected.

As mentioned during the presentation of the static semantic analysis, in Sect. 4.3, there are some errors that must be checked at run time. They include, for instance, references to non-existent objects, access to specialisation properties of non-specialised objects, and the existence in the action chain of birth and death actions for the same object. Additionally, the effect on the attributes of all actions belonging to the calling chain must not violate any integrity constraint. In case a state transition cannot take place, the effect of the actions executed up to that moment must be *undone*. So intermediate state changes are first done temporarily and only become permanent when the state transition is successfully completed. In our case, temporary states are hold on main memory and made permanent by storing them into the database.

6.1.5 Execution Model

Next, we present a sequential execution model of action calling that takes into account all aspects analysed above. First, we describe the execution model of an action included in a calling relation. We assume that a correct execution order was statically found and operations are executed according to it.

Algorithm 6.1.2 The execution model of an action belonging to an action chain is given by the next algorithm:

1. Check if an action instance with the same parameter values has already been executed in the object. If this is the case then set output parameter values and `return`.
2. Check the termination condition. If it is not satisfied then `abort("exceeded maximum number of occurrences")`
3. Check precondition. If it is not fulfilled then `abort("precondition does not hold")`
4. Do operations (assignments, action callings, ...). Determine the next (temporary) state. During the execution:

(a) Check modification conflicts. If a conflict is found then `abort("found modification conflict")`.

(b) Check other run-time errors. If detected then `abort("found run-time error")`.

□

As mentioned before, users simulates an event in the system by selecting an initial action. The execution of the inital action then starts the execution of the action chain, that is determined through the calling relation, and establishes a state transition in the system. The next algorithm completes the execution model of an initial action.

Algorithm 6.1.3 The exection of an initial action triggered by the occurrence of an event in the system is defined as follows:

1. Execute the initial action according to the previous algorithm.
2. In the temporary new state:
   (a) Evaluate derived attributes of participating objects.
   (b) Check integrity constraints in participating objects. If one or more constraints are violated then `abort("integrity constraint not fulfilled")`.
3. If the execution was not aborted make state changes permanent.

□

Since the valuation of derived attributes and the verification of integrity constraints depend on the global effect of the actions, they are done after all actions of the calling chain have been completed.

The execution model does not explicitly deal the occurrence of concurrent events coming from different places of the object society. However, the fact that objects are stored in a database allows users to animate simultaneously an object society. In this case, it must be guaranteed that the effect of each event on the object states is independent of all the others. For that, we isolate the execution of initial actions by using the transaction and lock mechanisms provided by the DBMS.

The next section presents how TROLL specifications are transformed into C++ following this execution model.

6.2 Transformation into C++

As we mentioned in the introduction to this chapter, in the synthesis phase a code generator transforms the TROLL specification into C++ code. C++ has been chosen as the target programming language because it supports the object-oriented paradigm and is widely available. Other object-oriented programming languages, such as Smalltalk and Java, could have been used as well, but the fact that the animator and the database interface are also implemented in C++ was a main reason for deciding on it. Once the code has been generated, it is compiled, linked into a shared library and then loaded dynamically in the animator. During the animation, users can navigate through the object society and select an action to be executed. The execution is then monitored by an execution manager that interacts with the library code that implements the action behaviour. In this section, we show how the execution model is implemented in the animator and how the specifications are transformed into C++. We focus on the main structure and abstract from implementation details. For this reason, we use an algorithm-like syntax when describing the code. A more exact transformation into C++ can be found in Appendix D. In what follows, we assume the reader has a basic background on C++ concepts. A comprehensive coverage of C++ language features based on the last ANSI/ISO standard can be found, for instance, in [Str97]. Next, we describe the requirements we have taken into account in the transformation from TROLL specifications into C++.

6.2.1 Code Requirements

There are several ways for representing the structure of TROLL specifications in the code. In our representation, we have considered the following requirements:

- *Use of object-oriented concepts:* Since C++ is a superset of C, it can also be used as a traditional programming language with no use of object-oriented concepts. Nevertheless, in the transformation, we avoid the use of mechanisms that are against the object-oriented paradigm.

- *Similar specification and code structures:* The TROLL class structure and object encapsulation must also be maintained in the implementation. This allows that changes in the specification do not require the compilation of all the code again, but only the compilation of the

corresponding C++ classes. Furthermore, as code reuse is an essential feature of the object-oriented paradigm, reuse of TROLL classes in other systems should also entail reuse of the C++ classes during the animation. So for instance, as in TROLL, global behaviour definitions must not be included in the local behaviour of the classes, components must not know about the composite classes and the definition of superclasses must be independent of their specialisations.

- *Efficiency:* Efficiency must also be considered if we want that users animate a specification in real time independently of the specification size and the number of objects in the system. This was also one of the reasons for using a code generator and not an interpreter for animating TROLL specifications. During the animation only the participating objects in a state transition are loaded into main memory. Moreover, component objects are not loaded automatically with the composite object but only if necessary.

- *Code reuse:* The fact that code is generated allows the possibility of using some of this code for the implementation of the system. Following an evolutionary approach, the generated code could be refined until obtaining the final implementation. This requires the generation of code that can be *legible*. For this reason, we separate, at most as possible, code belonging to the animation control from code that directly represents the behaviour of the actions.

We describe next the representation of the specifications in the code. We start with the description of the data types library.

6.2.2 Data Types Library

As we described in the last section, the valuation of attributes and variables during the execution of a state transition requires some run-time checks:

- All attributes and local variables used in the operations must have a value. There exist two cases in which an attribute is used and has not a value. The first one is when the attribute is declared as optional and its value has not be instantiated yet. The second one is when the object to which the attribute belongs does not still exists, i. e. it is just created during the current state transition. On the other hand, since

the instantiation of local variables may be conditioned, it is not always possible to statically determine whether a local variable has a value before it is used. So dynamic checks are necessary.

- Assignments to attributes and local variables must be consistent. That is, during a state transition, they cannot be assigned with different values.

In order to avoid to explicitly generate code for performing all these checks, we decided to encapsulate them in data type classes. Every TROLL predefined data type and constructor has been implemented in a C++ class. TROLL data type classes are named by the type name prefixed by a "t". Checks are performed whenever an operator of the data type is used. The next pattern describes, at an abstract level, how the class for a type <*type*> has been implemented:

```
class t<type>
  attributes
    value : <type>;
    has_value : bool;
  operators
                                                    assignment operator
    := (v : t<type>) do
      if v.has_value = false then
        throw exception("non-instantiated vble/attr was used")
      elif has_value = true then
        if value ≠ v.value then
          throw exception ("vble/attr assigned to different values")
        fi
      else
        value := v.value;
        has_value := true;
      fi;
    od;
    ...
end class
```

The attribute *value* will contain the value of the TROLL attribute or variable and the attribute *has_value* indicates whether a value has already been assigned. Operators have been overloaded for objects of the data type classes.

For instance, the assignment operator checks, first at all, if the parameter that represents the variable at the right side of the assignment has a value. If this is not the case an exception is thrown. The exception is caught by the execution manager that rejects then the state transition and reports the error to the user. If the parameter has a value, then it is checked if the attribute *value* has already been assigned. If this is the case and its value is different from the value to be assigned, then an exception is thrown. Finally, if *value* has not been assigned yet, it is assigned and the attribute *has_value* becomes true. Data type classes also include the redefinition of copy constructors and type conversions between constants of the data types and objects of the data type classes.

Table 6.1 shows how the TROLL types are represented in the code. The

TROLL **Type**	**C++ Type**	TROLL **Type**	**C++ Type**
nat	tnat	|class|	oid<class>
int	tint	enum	tstring
real	treal	record	record_class
money	tmoney	list(*type*)	tlist<t*type*>
char	tchar	set(*type*)	tset<t*type*>
string	tstring	bag(*type*)	tbag<t*type*>
bool	tbool	map(t_1,t_2)	tmap<tt_1,tt_2>
date	tdate		

Table 6.1: Representation from TROLL types in the C++ code

value of a natural number is defined as an integer. The class *tnat* throws an exception in case the value becomes negative. Object-valued types are represented by a class template whose attribute *value* is a pointer to an object of the corresponding class. The C++ template mechanism allows a type to be a parameter in the definition of a class. So a class template specifies a family of classes. As in the database, enumerations are represented by strings. For each record type a class is generated. The fields of the record are defined as attributes in the record class. Collection types are implemented with help of the container types provided by the C++ standard library. The representation in the C++ code of generic operators defined for each TROLL data type can be found in Appendix D.

6.2.3 Troll_Class

Each TROLL class is implemented in a C++ class. The C++ classes are derived from a base class, *Troll_Class*, that captures common attributes and services of the classes. The main attributes and services provided by the base class are defined as follows:

class *Troll_Class*
  attributes
    id : *list*(*record*(*name* : *string*, *param_val* : <*type*>, *class* : *string*));
    execution_type : *enum*(`BIRTH,UPD,DEATH,READ`);
    local_trace : *list*(*record*(*act* : *string*, *p* : *list*(*param*)));
    action_counter : *map*[*string*, *nat*];
    static action_limit : *nat*;
    static created_objects_counter : *map*[*string*, *nat*];
    static created_objects_limit : *nat*;
    ...
  functions
    set_execution_type(*ex_type* : *enum*(`BIRTH,UPD,DEATH,READ`));
    create_object();
    delete_object();
    ——————————— **to be implemented in the subclasses**
    virtual init_attributes();
    virtual read_attributes();
    virtual eval_der_attr();
    virtual check_constraints();
    virtual write_attributes();
    ...
end class

The attribute *id* contains the object's identity. It is defined by the global identification path of the object. Its structure is based on the identification structure given in Def. 5.2.1, which was used for describing the mapping of objects' states into the database. The first element of the identification list represents the identifier of the global object and the rest of elements correspond to the component and specialisation identifiers in the path. For global and component objects the record fields *name*, *param_val* and *class* represent the name, the parameter value, if required, and the class name respectively. For specialisations, the record field *class* represents the name

of the specialisation class. As we will describe later, a specialised object is represented in the code by several objects, one representing the properties of the base class and one for each of its specialisations. The identity of an object representing a specialisation aspect is built with the identification list of the base object plus a new element that includes the specialisation class. The attribute *execution_type* denotes the kind of participation of the object in the state transition. The value `BIRTH` means that the object is created in the state transition, i. e. a birth action is called in the object. The execution type `DEATH` denotes that a death action is called in the object. For the rest of actions, the execution type is `UPD`. The execution type `READ` means that the object is loaded on main memory because another object wants to know the value of its attributes. Since several actions of different types can be called on the same object during the state transition, the execution type of the object is determined by the type with a stronger meaning. For example, if an update and a death action are called in an object, then the execution type of the object is `DEATH`. The execution type determines the operations to be performed on the object to follow the execution model. For instance, in the case of `READ`, it is not necessary to evaluate derived attributes or integrity constraints in the new object state because its state has not changed. The execution type also determines the communication with the database. Since an object with execution type `BIRTH` does not exist in the previous state, its attributes are not retrieved from the database. During the database commit, the persistence layer receives the additional command to create the object in the database. The execution type is set by the function *set_execution_type*. As mentioned in the previous section, the execution of an action instance occurs at most once during the state transition. That is, several callings to the same action instance lead to a unique execution of the action instance in the state transition. So whenever an action is called, it must be assured that the action, with also the same input parameter values, has not been called before. To this end, it is necessary to store for each object which action instances have already been called. The attribute *local_trace* stores in each object the name and parameter values of each action called during the state transition. The rest of attributes in the *Troll_Class* definition are concerned with the termination of state transitions. The attribute *action_counter* contains for each action in the object, the number of action instances that have been executed. This number may not exceed the value of the attribute *action_limit*. The keyword *static* preceding the attribute means that the attribute value is shared among all objects of the class. The attribute *created_objects_counter*

contains for each TROLL class, the number of objects created during the state transition. The maximum number of objects created of a class is given by the attribute *created_objects_limit*. The boundary values, *action_limit* and *created_objects_limit*, can be set by the users during the animation. Besides the function that sets the execution type, other common functions provided by the *Troll_class* include functions for communicating with the database such as *create_object* and *delete_object*, and functions whose implementation is defined in the subclasses. The keyword *virtual* preceding a service means that the service may be overridden in derived classes. When an object is created, the function *init_attributes* instantiates attributes declared as *initialised* with the values given in the specification. *read_attributes* retrieves from the database the attribute values in the current state. Once all actions of the action chain have been executed, the functions *eval_der_attr* and *check_constraints* evaluate derived attributes and integrity constraints in the temporary new state respectively. After the state transition has been successfully performed, *write_attributes* stores the new values of the attributes in the database.

6.2.4 TROLL Classes

Once the class definition of the base class has been presented, we describe the structure of the C++ class generated for each TROLL class. As in TROLL, C++ classes are composed of a class signature definition and a behaviour definition that includes the implementation of each service provided by the class. So for a TROLL class, we directly map its signature into the C++ class definition and its behaviour definition into the function implementation part of the C++ class. Next, we present the transformation of the TROLL class signature into the C++ class definition.

Class Definition

A class definition consists of the declaration of attributes, components, specialisations and actions.

- **Attributes**

During the state transition, there are two values to be managed for each attribute: the value in the current state and the value in the temporary new state. Attribute values used in the operations do always reference to

the current state whereas values assigned to attributes represent values in the new state. Thus the C++ class contains two attribute declarations for each attribute declaration in the TROLL class. These attributes will contain the value of the TROLL attribute in both states. The data type of the C++ attributes is defined according to table 6.1. After all actions have been performed, derived attributes and integrity constraints must be evaluated in the new state, i. e. using the new attribute values. Here, we must consider that the values in the new state of attributes that have not been affected by the state transition correspond to the values they have in the previous state. To this end, a C++ function is generated for each TROLL attribute. If an attribute has been assigned in the new state, the function returns the new attribute value, otherwise it returns the value in the previous state. This function is used in the valuation of derived attributes and constraints checking whenever an attribute is referenced. The new attribute values are then made permanent by storing them in the database.

- **Components**

Components are not explicitly supported in C++. Thus they are mapped into attributes. A single component is represented by a reference to an object of the component class. A parametrised component is represented by a map that has as domain the parameter values and as range references to the component objects. Access to components will be explained later.

- **Specialisations**

As in components, TROLL specialisations cannot be directly represented in C++. Although the specialisation relationship in TROLL and the inheritance mechanism in C++ have similar concepts there are basic differences between them. In TROLL, the behaviour definition of an action of the base class in a specialisation extends the behaviour of the action but does not override it. That is, if the action is called in an object that is specialised, then the behaviour of the action defined in both the specialisation and the base class must be executed. This can be simulated in C++ by explicitly calling in the specialisation the action in the base class.
A main difference lies on the fact that, in TROLL, objects are declared of the base class and are specialised at the moment of its creation if the specialisation condition is satisfied. Whenever we have an object declared of the base class we can ask if the object has a specialisation (using the operator

isA) and then call in the object an action defined in its specialisation. In C++, besides there is not any explicit specialisation condition, objects and references to objects must be declared of a determined class, either of the base class or of the derived class. An object declared of the base class can never have access to attributes and actions defined in the derived class. For instance, in the CATC example presented in the introduction to TROLL (see page 51) we declare parametrised objects of class `User`. A user may be specialised at the moment of its creation in `Operator` and `Staff`. In the global behaviour specification, we then model the behaviour of `Operator` actions for objects declared as `Users` that have also the `Operator` aspect. In C++, for specifying the behaviour of the `Operator` actions we should have explicitly declared the objects as `Operators`. A possible solution would consist in delaying the declaration of the C++ objects until the moment the birth action is called. The specialisation condition would be first at all evaluated and then an object of the corresponding class, either of the base class or of the derived class, would be declared. The birth action would then be called in the object. Even using this solution, a main problem is that, in TROLL, objects may simultaneously have several specialisations. For instance, if a TROLL object has two specialisation aspects then we have to declare a C++ object that has both aspects as well. This can be done by using the multiple inheritance mechanism of C++. A class should be generated that is derived from the classes that represent the specialisation aspects. Consider in TROLL a class `Person` that has two specialisation classes `Student` and `Worker` whose specialisation conditions are not disjoint. That is, it is possible that an object person is simultaneously a student and a worker. In C++, we need to generate four classes: one for the class `Person`, one for each aspect, `Worker` and `Student`, and one for the combination of both aspects, `Worker+Student`. The C++ object is then declared of one of these classes depending on the evaluation of the specialisation conditions. Note that a derived class must be generated for each possible combination of the specialisation aspects. Because of the complexity of this solution, we decided not to make use of the inheritance mechanism for representing TROLL specialisations in C++. In the code, each specialisation aspect is represented by a different object. So in the example above, a person that is both a worker and a student is represented by three objects (`Person`, `Worker` and `Student`). When the birth action is called, an object of the TROLL base class is always created. Then, the specialisation conditions are evaluated and the corresponding specialisation objects are created. If an action called in a *base* object is extended

in specialisations, then the action is called in the specialisation objects too. Actions defined exclusively in specialisation classes will be executed only in the corresponding specialisation objects. Note that there is no problem for determining to which specialisation an action belongs, because in the TROLL specification, the specialisation aspect must be explicitly written in the action calling. As we will describe later, all the code generated for dealing the specialisation mechanism is encapsulated in one function.
As a TROLL specialisation can refer to attributes, components and actions declared in the signature of the superclass, an attribute in the C++ class representing the specialisation will contain a reference to the superclass object. The access to the signature elements of the superclass will be done through this attribute.

- **Actions**

Each action declared in the signature of the TROLL class is directly mapped to a function declaration in the C++ class. We must also consider that, in some cases, a TROLL class can define the behaviour of actions that are not explicitly declared in the class. A specialisation class can extend the behaviour of actions whose declaration is inherited from the superclass (or if the superclass is in turn a specialisation, from the superclass of the superclass, etc.). In the same way, a composite class can extend the behaviour of actions declared in its components. These actions must also be declared in the C++ class definition. Note that if the behaviour of an action belonging to a multiple component is extended in the composite class, the parameter identifiers of the component can be used in the definition of the action. For instance, in the example 4.3.2 on page 77, the object class `A` extends the behaviour of the component action `Cs(n).actC(p1)`. The parameter `n`, that represents the parameter value of the component, is visible in the definition of the action. For this, we include the parameter identifier as an input argument in the C++ function declaration. So the behaviour definition of `actC` in the class `A` is represented in the C++ class by the function `Cs_actC(n,p1)`. In the C++ functions, parameters are passed *by reference*. Input parameters are declared *const* to indicate that the called function will no alter their values. Since output parameters are passed by reference their values can be modified by the called function.
Additionally, the functions declared as *virtual* in the base class, $Troll\_Class$, that have an implementation in the class must also be declared in the class definition.

As a summary of the mapping rules cited above, the next pattern shows how the signature of a TROLL class, <*class_name*>, is represented in the C++ class definition:

class <*class_name*> **inherits from** *Troll_Class*

attributes

for each attribute declaration in the TROLL class
<*attr*> : <*type*>$_{C++}$;
<*attr*>_*new* : <*type*>$_{C++}$;
...
for each component declaration in the TROLL class
if single
<*comp*> : *ref* <*comp_class*>;
if multiple
<*comp*> : *map*[<*type*>$_{C++}$, *ref* <*comp_class*>];
...
if the TROLL class is a specialisation
superclass : *ref* <*superclass*>;

functions

for each action declaration in the TROLL class
<*action*>(<p_1> : <*type*$_1$>$_{C++}$, ..., <p_n> : <*type*$_n$>$_{C++}$);
...
If the TROLL class is a specialisation,
for each extended action of the superclass
<*superclass_act*>(<p_1> : <*type*$_1$>$_{C++}$, ..., <p_n> : <*type*$_n$>$_{C++}$);
...
If the TROLL class has components,
for each extended action of the components
<*comp_act*>(<i_1> : <t_1>$_{C++}$, ..., <i_m> : <t_m>$_{C++}$,
<p_1> : <t_1>$_{C++}$, ..., <p_n> : <t_n>$_{C++}$);
...
for each attribute return its value in the new state
*get_*<*attr*>*_new*();
...
common functions that are implemented in the class
init_attributes();
read_attributes();
eval_der_attr();

check_constraints();
write_attributes();
end class

Once the transformation of the TROLL class signature into the C++ class definition has been presented, we show how the behaviour of TROLL actions is mapped into the implementation of the C++ class.

Behaviour Definition

Each TROLL action is implemented by a function in the C++ class. The implementation is based on the execution model described in the previous section (see algorithm 6.1.2 on page 117). In the implementation, actions call directly other actions defined in the same or in a different object. That is, before the calling to an action, there is not any intermediate function that controls the execution. Thus required checks such as checking that the action was not executed before are implemented in the generated functions. Next, we describe the implementation of each action in the code. To make easier the explanation, we distinguish between control code and code that represents the local behaviour of the action.

- **Control Code**

The control code generated for each action performs the following tasks:

- Check that the action was not executed before.
- Check that the action chain does not include several birth/death actions.
- Check that the termination conditions are not violated
- Store the action instance in the local trace of the object.
- Call the functions that manage the calling to action extensions and global interactions.

Note that the control code is also necessary for actions that do not have an explicit local behaviour definition in the specification. Although most of the actions have a local behaviour definition there are cases where the definition is not necessary. For instance, birth and death actions have already an inherent

behaviour, namely the object's creation and destruction respectively, and may not require any extra behaviour definition. It is also possible that the action's behaviour is not defined locally but in the global behaviour part, or in specialisation and composite classes. In any case, since actions are always called locally, each action declared in a TROLL class must have a function implementation in the corresponding C++ class that includes at least the control code. On the other hand, functions representing actions extensions in specialisation and composite classes do not require any control code because it is already included in the local action definition.

The next pattern describes, at an abstract level, the control code generated in the C++ function for a TROLL action, $<action>$, of a class, $<class>$, with input parameters, $<i_1>, \ldots, <i_n>$, and output parameters, $<o_1>, \ldots, <o_m>$:

$<class> :: <action>(<i_1> : <type_{i_1}>_{C++}, \ldots, <i_n> : <type_{i_n}>_{C++},$
$<o_1> : <type_{o_1}>_{C++}, \ldots, <o_m> : <type_{o_m}>_{C++})$

do

_____ was the action already executed?
if *action_in_trace*(“<*action*>”,<i_1>, . . . , <i_n>) **then**
assign_output_values(<o_1>, . . . , <o_m>);
return;
fi;

_____ only for birth and death actions,
_____ was another birth or death action already executed?
check_birth_death_conflicts();

_____ only for birth actions, max. number of created objects?
check_and_incr_created_objects_counter(“<*class*>”);

_____ max. number of occurrences of an action?
check_and_incr_action_counter(“<*action*>”);
_____ update local trace
to_local_trace(“<*action*>”, <i_1>, . . . , <i_n>);
_____ execute the TROLL behaviour code
. . .

_____ update output parameter values in local trace
to_local_trace(“<*action*>”, <o_1>, . . . , <o_m>);
_____ call possible extensions and global interactions

call_extensions(“<*action*>”, <i_1>, . . . , <i_n>, <o_1>, . . . , <o_m>);
call_global_interactions(“<*action*>”, <i_1>, . . . , <i_n>, <o_1>, . . . , <o_m>);
od;

First, it is checked if an action with the same name and input parameter values is already in the local trace. If this is the case, then the output parameters of the action are assigned and the function returns. That is, the action instance is not executed because it was executed before. In case the action is declared as a birth or death action, it is then checked that no birth or death action is in the local trace. Next, termination conditions are evaluated. For a birth action, it is checked that the number of objects created of the class up to that moment (stored in the attribute of the base class, *created_objecs_counter*[“<*class*>”]) has not reached the maximum allowed (attribute *create_objects_limit* in the base class). If this is not the case, the counter is incremented. Similarly, it is checked that the number of times the action has been executed (attribute *action_counter*[“<*action*>”]) has not reached the boundary (attribute *action_limit*). If one of these checks is not fulfilled an exception is thrown which returns automatically the control to the execution manager that reports the error to the animator users. If all the checks have been satisfied the action name together with the input parameters are stored in the local trace. The behaviour code defined in the specification is then executed. We describe this code below. After the behaviour code has been executed, the output parameter values are stored in the local trace. At last, two special functions: *call_extensions* and *call_global_interactions* are called. These functions are also generated and manage the call to other behaviour extensions of the action. We have encapsulated these calls in functions outside the classes to follow the encapsulation principle of TROLL. As in TROLL, a class is defined independently whether it is a component of other classes, is specialised or has a global behaviour. This allows to reuse the code in the same way TROLL specifications can be reused. We will describe both functions later.

Next, we describe how the behaviour definition of an action specified in the TROLL class is represented in the C++ function.

- **Behaviour Code**

The behaviour definition for a TROLL action consists in the definition of action preconditions, declaration of local variables and the definition of the operations describing the effect of the action on the system. Operations are

classified in assignments, action callings, conditionals and multicallings. In what follows, we concentrate on the structure of the generated code and refer the reader to Appendix D for further details. The next pattern extends the previous one with the transformation of the behavior definition of the TROLL action into the C++ function. The pattern illustrates the structure of the generated code and give some examples:

$<class>$ :: $<action>(<i_1> : <type_{i_1}>_{C++}, \ldots, <i_n> : <type_{i_n}>_{C++},$
$<o_1> : <type_{o_1}>_{C++}, \ldots, <o_m> : <type_{o_m}>_{C++})$

do

control code

...

check precondition

if *not* $<prec>$ **then**
  throw *exception*("$<prec>$ does not hold");
fi;

in birth actions, assign initialised attributes

init_attributes();

declare local variables

$<v_1> : <type_{v_1}>_{C++}, \ldots, <v_k> : <type_{v_k}>_{C++}$;

do operations

examples

assignment to attributes

$<attr>\_new := <attr> + <i_1>$;

access to the signature of components

get_comp($<comp>$, $<pval>$,"$<comp>$","$<comp\_class>$", $<action\_type>$);
$<comp>[<pval>] \rightarrow <comp\_action>(<p_1>, \ldots, <p_n>)$;

access to the signature of the superclass

$<o_1> := superclass \rightarrow <attr>$;

conditionals

if $<formula>$ **then**
  ...
fi;

multicalling to actions of components

get_all_comps($<comp>$,"$<comp>$","$<comp\_class>$", $<action\_type>$);
$<x> : <type_{cmp\_param}>$;
for $<x> := dom(<comp>).first()$ **to** $dom(<comp>).last()$ **do**
  $<comp>[<x>] \rightarrow <comp\_action>(<p_1>, \ldots, <p_n>)$;

```
    od;
    ...
    ------------------------------------------ control code
    ...
od;
```

If the action precondition is not fulfilled, an exception is thrown that stops the execution of the whole state transition and gives the control back to the execution manager. The reason for the rejection of the state transition is then reported to the users. Birth actions initialise the values of attributes declared as *initialised* by calling the action *init_attributes*. Local variables are then defined. Next, operations are executed according to the execution order determined before the code generation. Assignments to attributes are always performed on the new state (represented by the C++ attribute <*attr*>_*new*). On the other hand, attribute values used at the right part of the assignments corresponds to the current state (represented by the C++ attribute <*attr*>). As we described in the presentation of the data types library, conflicts in attribute and variable assignments as well as the use in statements of non-instantiated attributes and variables are automatically detected in the type classes. Component objects are not created on main memory together with the compound objects, but only when they are required. Whenever a component is required, the function *get_comp* checks if the corresponding element (<*pval*>) in the components map (<*comp*>) already contains a reference to the component. If not, *get_comp* builds the global identity of the component object and requests the execution manager to create the component object on main memory and to return a reference to it. The global identity of the component is built with the identity of the composite plus the local identity of the component, that is passed as argument in the function. As we will see later, if the component already exists in the database, its attributes values are retrieved from the database at the moment the object is created on main memory. Additionally, the kind of action to be executed on the component, `BIRTH, UPD` or `DEATH`, is passed in the argument <*action_type*>. If the composite wants just to read the value of an attribute of the component, then the value `READ` is passed in the argument. The execution type of the component is set according to this argument. Once the reference to the required component is contained in the components map, the action is directly called in the component. For

a single component the procedure is the same but no parameter values are managed, and the component is represented as a simple type attribute and not a map. As can be seen in the examples, the access to the signature of the superclass is done through the attribute *superclass*. Unlike components, the attribute *superclass* is always instantiated. During the execution, the object that represents the properties of the base class and the objects representing each specialisation aspect are created together. A reference to the base object is then assigned to the attribute *superclass* of each specialisation object. The translation of conditional statements with no quantified conditions into C++ is straightforward. A quantified formula requires the use of a loop in order to evaluate the formula for each possible instance of the quantified variable. If the formula is universally quantified, it is evaluated for every possible variable instance until it becomes false or is satisfied for all elements in the values set. On the other hand, if the formula is existentially quantified the procedure is the opposite. The loop finishes when a variable instance is found for which the formula holds or when the formula is not satisfied for any instance of the values set. In both cases, a bool variable contains the final result of the evaluation. Note that loops always end because quantified variables must always have a finite set of values. The last example is a multicalling to actions of components. As quantified formulas, multicallings are implemented using loops. The function *get_all_comps* is similar to the function *get_comp*. The difference is that *get_all_comps* loads all components existing in the database. So no parameter value is passed as argument.

Additional Functions in the C++ Classes

Besides the generation of a function for each TROLL action in the C++ class, some special functions are generated too. For each attribute in the TROLL class, a function is generated returning the attribute value in the new state. The rest of functions are declared in the C++ base class, $Troll\_Class$, and are implemented, if required, in each C++ class. We do not show here the generated code for each of these functions but just describe their functionalities:

- *init_attributes* : This function initialises all *initialised* attributes of the class at the moment of the object's birth. Since the attributes will be first visible in the next state, initial values are assigned to the C++ attributes representing the new state (<*attr*>_*new*).

- *read_attributes* : This function retrieves the current attribute values from the database and assigns them to the corresponding attributes in the C++ object. This action is called when the object is created in main memory and its execution type is different to `BIRTH`.

- *get_<attr>_new*: This function returns the value in the new state of the attribute *<attr>*. If the attribute has been modified during the state transition, the function returns the new value (*<attr>_new*) otherwise the function returns the value in the previous state (*<attr>*). This function is used by the next two functions.

- *eval_der_attr* : This function evaluates derivation terms of derived attributes in the new state and assign them to the corresponding attributes.

- *check_constraints* : This function evaluates integrity constraints in the new state. The evaluation of integrity constraints are similar to the evaluation of action preconditions. If the negation of a constraint is satisfied an exception is thrown an the state transition does not take place.*Initially* constraints are additionally evaluated if the object has been created during the state transition (its execution type is `BIRTH`).

- *write_attributes* : This function stores the values of the attributes that have been modified into the database. By making this, state changes are done permanent.

6.2.5 Other Generated Functions

Besides the generation of a C++ class for each TROLL class defined in the specification, other functions are generated. They include:

- *call_extensions*:

This function is called after the execution of each action in the objects and manages the specialisation relations between objects and also the call to action extensions defined in compound objects. As indicated before, TROLL specialised objects are represented by several C++ objects: one representing the base properties and one for each specialisation aspect. This means that the specialisation relations between these objects must be explictly managed in the code. On the one hand, if a birth action has been called in an object

that may be specialised, the specialisation conditions are evaluated. If one or more specialisation conditions are satisfied *call_extensions* requests the execution manager to create the corresponding specialisation objects on main memory and calls, if defined, the birth action in these objects. On the other hand, if an update or death action has been called in an object that is specialised, and the action is defined in the specialisations, the action is called in the corresponding specialisation objects. Similarly, if an action is called in a component object and is defined in the composite, the corresponding action in the composite is also called. Note that to know if an object may be specialised is determined by its class. On the other hand, since a class may be used in several object and component declarations, to know if an action called in an object is extended in a compound object is determined not by the object class but by the global identification path of the object.

- *call_global_interactions*:

This function implements the global behaviour definition of actions. As the previous function, this function is called after the execution of each action. The function checks if a global behaviour definition exists for the action and the identification path of the object in which it is called. In this case, the corresponding behaviour code is executed. The structure of this code is similar to the structure of the code generated in the C++ classes for the local behaviour of the actions. When calling an action in other object, the identity of the object is built and a reference to the object is requested to the execution manager. The action is then directly called in the object.

- *call_initial_action*:

This function represents the interface between the animator and the generated code. The function receives as arguments an object's identity, an action name and a list of parameter values and then calls the corresponding action in the code. The procedure is as follows. First, the function declares variables for the action parameters. The parameter values passed in the argument are then assigned to the variables representing input parameters of the action. Next, the function creates the corresponding C++ object through the execution manager and calls the action, with the parameter variables as arguments, in the object. Finally, once the action has been executed, the values of the output parameters are returned in an argument. If an output parameter has not been instantiated, an exception is thrown and the transition is rejected.

The generation and an example of these functions can be found in Appendix D.

6.2.6 Execution Manager

The execution manager monitors the execution of state transitions in the system. In the animator, users navigate through the object society and observe the current state of the objects, that is stored in the database. Users simulate an event in the system by selecting an initial action in an object to be executed. If required, values for the input parameters of the action are introduced. To create a new object instance, users must introduce the identity of the new object. The initial action then corresponds to the birth action of the object. If several birth actions are defined in the class of the object, users can select one of them. Once the initial action has been selected, the animator gives the execution manager the object identity, the action name and the values of the input parameters. The execution of the action follows the execution model given in the previous section (algorithm 6.1.3 on page 118). Firstly, the initial action is called through the function *call_initial_action* defined before. All actions of the action chain determined by the calling relation are then executed in the objects. Figure 6.1 illustrates this phase. The execution manager handles the creation of objects on main memory and stores in a vector references to them. When an object is required, because an action in the object must be called or because the value of one of its attributes must be used, a reference to the object is requested to the execution manager. The identity of the object together with its execution type (either `BIRTH, DEATH, UPD` or `READ`) are passed as arguments. The execution manager checks if the object already exists on main memory by looking for an object with the same identity in the vector of object references. If the object is not found, the execution manager creates the object on main memory and introduces a reference to the new object in the vector. In both cases, the execution manager calls the function *set_execution_type* in the object and returns the requested object reference. If the object was not in main memory before and its execution type is different to `BIRTH`, i. e. the object already exists in the object society, the function *set_execution_type* calls the function *read_attributes* that retrieves the attribute values of the object from the database. Errors concerned with the existence or non-existence of objects in the database are detected here. For instance, if the execution type of the object is `BIRTH` and the object already exists in the database. Another

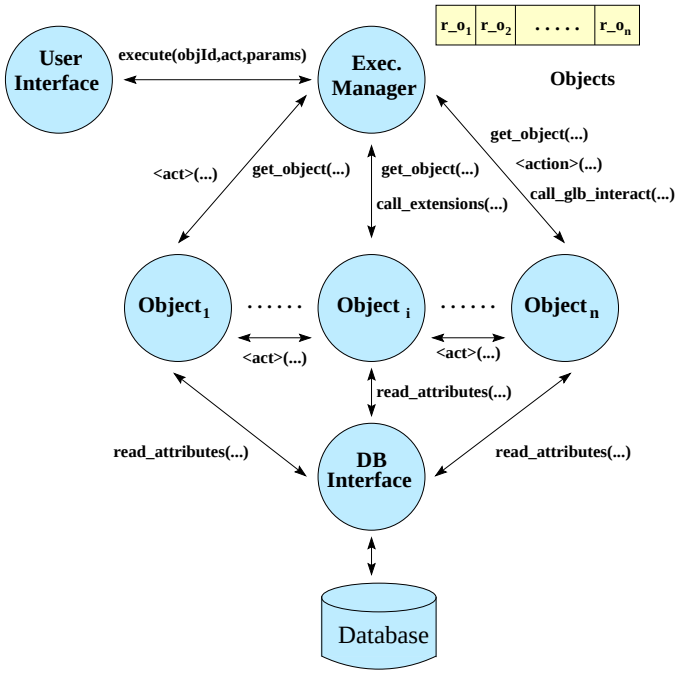


Figure 6.1: 1st Phase: Execution

error is, for example, when the execution type is `UPD` and the object does not exist in the database. In these cases, an exception is thrown and the state transition is rejected. For objects with specialisation aspects, the corresponding specialisation objects are also created in main memory with the same execution type. A reference to the base object is then assigned to the attribute *superclass* of each specialisation object. The procedure of action calling was described previously. Actions may call actions in the same object and in components. Action extensions in specialised and composite objects are called by the function *call_extensions* and global interactions are described in the function *call_global_interactions*. The call to these functions is handled by the execution manager. If a run-time error or the violation of an action precondition is detected the state transition is rejected and the cause is reported to the user. All objects created in main memory are then destroyed by the execution manager. As the database was not modified, the objects' states remain the same as before the calling to the initial action.

Once all actions in the objects have been executed, the next phase con-

sists in the valuation of derived attributes and the check of integrity constraints in the new state. Figure 6.2 describes this phase. The execution

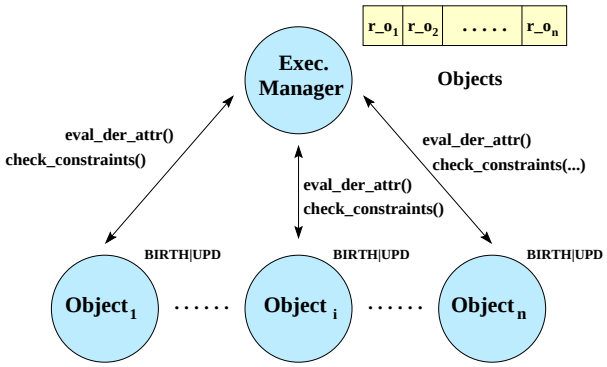


Figure 6.2: 2nd Phase: Valuation of Derived Attributes and Contraints

manager calls *eval_der_attributes* and *check_constraints* only in the objects whose execution type is either `BIRTH` or `UPD`, i. e. in the objects whose state may have been changed. Actually, only derived attributes and integrity constraints whose valuations depend on attributes that have been changed in the state transition need to be evaluated. For simplicity, the current version of the animator does not support this optimisation. As indicated previously, *eval_der_attributes* and *check_constraints* use the function generated for each attribute, *get_<attr>_new*, to access to the values of the attributes in the new state. So if the value of an attribute has not been changed in the state transition, its value in the previous state is used. For objects with execution type `BIRTH`, initially constraints are also evaluated. If a constraint is not satisfied, an exception is thrown. As in the first execution phase, the state transition is then rejected and the objects created on main memory are destroyed.

After the valuation of derived attributes and the check of integrity constraints in the new state, the last phase consists in making the state changes permanent in the database. This is illustrated in Fig. 6.3. The execution manager calls in the objects the functions that interact with the database interface according to the execution type. Objects with execution type `BIRTH` are created in the database by the function *create_object*. The function *write_attributes* stores the new value of the attributes for objects whose execution type is either `BIRTH` or `UPD`. Objects whose execution type is `DEATH`

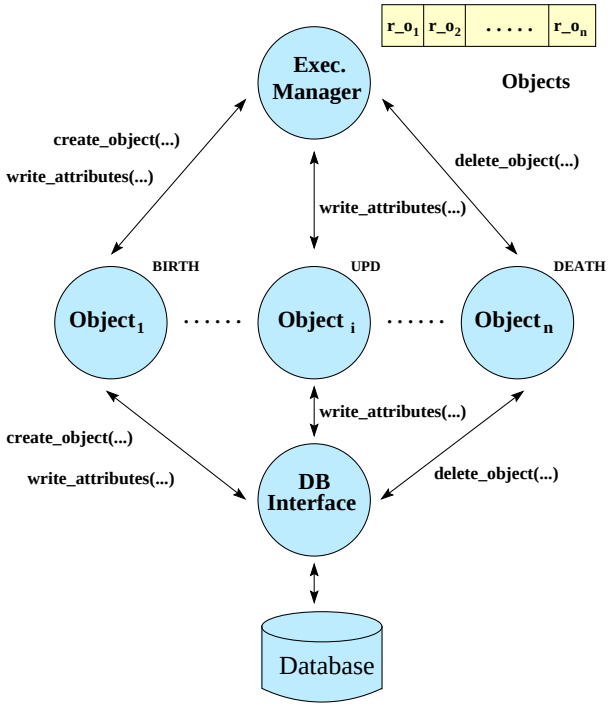


Figure 6.3: 3rd Phase: Database Update

are deleted from the database.

After the database update, the state transition is completely finished. Objects on main memory are destroyed and the values of the output parameters defined in the initial action are returned to the animator interface.

6.3 Summary

In this chapter, we have first analysed the behaviour of TROLL specifications and its execution, at the same abstraction level, using a sequential programming language. The execution of an initial action and all actions determined by the calling relation establishes a state transition in the system. In the sequential execution of a state transition the following aspects must be taken into account:

- Conceptually, all operations defined in the actions occur in parallel. For

the implementation of the specification in a sequential programming language a correct execution order, free of data flow dependencies, must be determined.

- Assignments to attributes and variables must be consistent. Since operations may be conditioned, run-time checks that guarantee the consistency are necessary.

- Actions are executed following an *all or nothing* approach. If an action cannot take place, then the effect on the attributes of all actions executed up to that moment must be undone.

- The termination of the state transition must be assured. For that, the number of instances of an action and the number of objects created of a class must be limited.

- No integrity constraint in the objects' states may be violated. Since the valuation of integrity constraints depends on the global effect of the actions, they must be evaluated after all actions have been executed.

- All actions involved in a calling relation must be executed in isolation. That is assured by the transaction and lock mechanisms of the DBMS.

According to these aspects, an execution model has been given. The implementation of the execution model in the animator and the structure of the C++ code generated from the specifications have been presented. The structure of the generated code is similar to the structure of the specification. So reuse of TROLL classes also allows reuse of the corresponding C++ classes in the animator. The transformation rules are summarised as follows:

- Each TROLL class is mapped into a C++ class. These classes are derived from a base class that provides common attributes and services to the classes.

- Attributes of the TROLL classes are mapped into two attributes in the C++ classes. These attributes contain the values of the TROLL attributes in the current and in the new states.

- Components are represented by attributes in the C++ classes that refer to the component objects.

- Specialised objects are represented by several objects: one representing the properties of the base class and one for each of the specialisation aspects.
- Each action in the TROLL classes is mapped into a function in the C++ classes.
- The termination conditions are checked in the C++ functions.
- The instantiation of initialised attributes, evaluation of derived attributes, constraints checking and the retrieve/update of attributes from the database are performed by functions in the C++ classes.
- The call to action extensions in specialisation and compound classes as well as the implementation of global interactions are defined in functions outside the classes.
- Inconsistencies in attribute and variable assignments are detected in the data type classes.
- Exception handling is the mechanism used for aborting the execution if any of the checks detects that the state transition cannot take place. In this case an exception is thrown and the cause is reported to the users.

An execution manager monitors the execution and handles the creation of objects on main memory. State transitions are executed in three phases. In the first phase, the initial action and all actions determined by the calling relation are executed. The evaluation of derived attributes and the check of integrity constraints in the new state are performed in the second phase. In a last phase, the new state of the objects is done permanent in the database.

Chapter 7

The TROLL Workbench

In this chapter, we describe the software development environment that has been developed for modelling and validating conceptual models specified in TROLL. The TROLL workbench is a set of software tools designed to facilitate the tasks of managing, editing, checking, documenting and validating TROLL specifications. For the validation of the models, an executable prototype is automatically generated from the models according to the steps discussed in the preceding chapters. In this chapter, we first give an overview of the tools and describe the workbench architecture. We then describe each tool contained in the workbench by example.

7.1 Architecture

The TROLL workbench is a collection of tools which support the development of formal specifications using the specification language TROLL. The tools can be used stand-alone from the command line and also together trough a tool that manages their integrated use. This is illustrated in Fig. 7.1. Currently, the workbench includes the following tools:

- *trlbench*: A TROLL projects management tool. *trlbench* is the graphical front-end of the workbench and provides a common interface to the other TROLL tools. Here, users can manage specification projects (create new projects, add TROLL files to a project, etc.) and invoke the other TROLL tools.

- *trlgred*: A OMTROLL graphical editor. *trlgred* provides users with

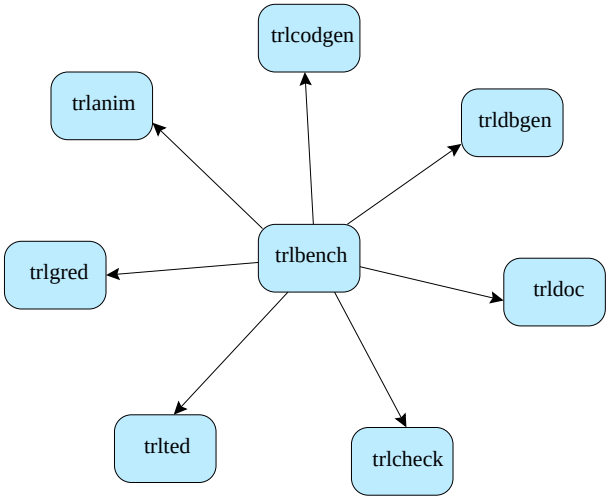


Figure 7.1: The TROLL Workbench

different diagram editors for modelling the system. Graphical models can be automatically translated into textual TROLL. *trlgred* allows also a partial import from TROLL specifications into OMTROLL.

- *trlted*: A TROLL language mode for the (X)Emacs editors. *trlted* adds new functionalities to these editors such as searching for specification components through the project files, automatic indentation and different colours and font styles for reserved words.

- *trlcheck*: A TROLL parser and static semantics checker. *trlcheck* can be embedded into the textual editor.

- *trldoc*: A TROLL documentation tool. *trldoc* generates HTML code from TROLL specifications. Using a HTML browser, the generated code allows users hypertext navigation through the specification and the introduction of informal comments to document the model. Additionally, *trldoc* is a pretty printer.

- *trldbgen*: A TROLL database generator. *trldbgen* generates database schemas from TROLL specifications. The databases contain the object instances created on animation time.

- *trlcodgen*: A TROLL-C++ code generator. *trlcodgen* transforms TROLL specifications into C++ programs. The resulting code is then compiled and dynamically loaded in the animator.

- *trlanim*: A TROLL animator. *trlanim* allows users to navigate through the object interfaces and to simulate occurrences of events in the system to see if the model's dynamics meet the user requirements.

Fig. 7.2 depicts the architecture of the workbench. The system can be mod-

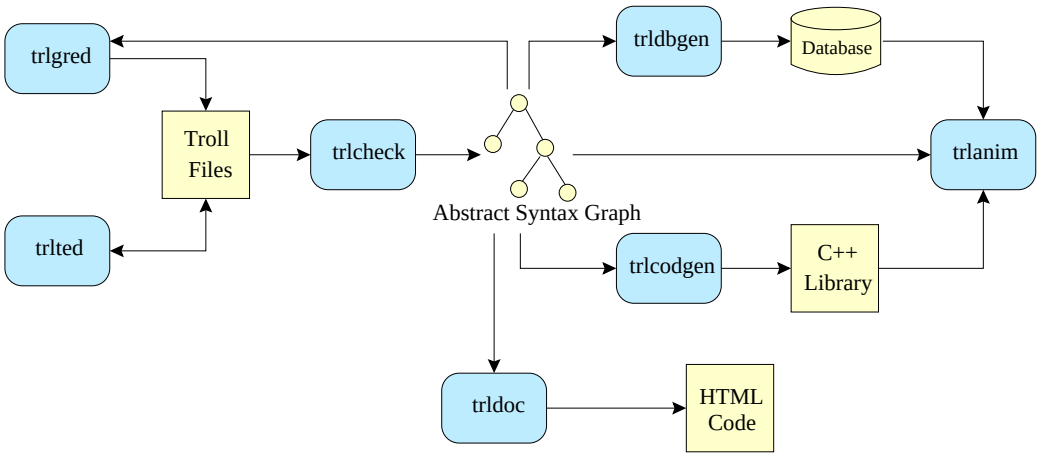


Figure 7.2: Architecture of the TROLL Workbench

elled using the graphical and textual editors. The specification is then stored in one or several ASCII files. The syntax and static semantics of the specification are checked by *trlcheck*. As a result, an abstract syntax graph is created. The graph is used in the workbench as the internal representation of the specification. All the remaining tools will access the information contained in the graph through an interface. *trldoc* generates HTML code from the specification. The HTML code includes also links that allow hypertext navigation through the specification. *trldbgen* reads the static structure of the specification contained in the graph and generates the corresponding database schema that will contain the state of the objects during the animation. *trlcodgen* generates C++ code from the specification. Once the code has been generated, *trlcodgen* calls the C++ compiler that compiles and links the C++ code into a dynamic library. Users can then animate

the specification using the TROLL animator, *trlanim*. The animator uses the syntax graph to obtain information about the specification, the database to retrieve and update the state of the objects created during the animation and the C++ library to execute the dynamics of the objects.

The workbench has been designed to facilitate its modifiability and extensibility with new tools. Since the functionalities of the workbench have been implemented in separate tools and the access to the data structures is performed through abstract interfaces hiding implementation details, changes, for instance, in the language syntax or in the graph structure, are easily localised and do not entail changes in all the tools. New tools can easily make use of the output of existing tools. The graph interface provides new tools with an easy and quick access to any information about the specification. New tools can be invoked and used together with existing ones through *trlbench*.

Implementation Issues

The TROLL workbench runs currently on Unix/Linux operating systems and may be easily ported to MS Windows using the `Cygwin`[1] tools. The workbench has been developed using exclusively free software tools. In the implementation, we have used C++ as the basic programming language and Tcl/Tk for the user interfaces. A prototype version of the workbench can be downloaded from the project's web site[2]. The source code is distributed under the terms of the GNU General Public License as published by the Free Software Foundation[3]. The source code distribution is autoconfiguring. The configuration scripts conform to the GNU standards and have been generated using the GNU development tools: `Autoconf`, `Automake` and `Libtool`.

7.2 Tools Description

In this section, we describe the main functionalities of each tool contained in the TROLL workbench. For the description of the tools, we use the example

[1] `Cygwin` is a UNIX-compatibility library that can be used to port UNIX software to Windows operating systems. More information about `Cygwin` can be found at http://sourceware.cygnus.com/cygwin/.

[2] The home page of the TROLL workbench is located at http://www.cs.tu-bs.de/idb/projects/troll-work.html

[3] The home site of the FSF is located at http://www.gnu.org

of the CATC system that was presented in the introduction to (OM)TROLL in Chapter 3. The complete specification of the example can be found in Appendix B.

7.2.1 *trlbench* - TROLL Projects Management Tool

trlbench is the graphical front-end of the workbench. This tool supports the management of specification projects and provides an easy access to the TROLL tools. In this way, users do not need to explicitly call each tool in the required sequence and with the required parameters and may abstract from the workbench internals. For each project, *trlbench* stores internally the files required for animating the specification such as the syntax graph and the C++ library. The invocation of tools that generate the required files is automatically handled. In *trlbench*, users may also configure the TROLL tools and the calling to external tools, i. e. the text editor and the HTML browser. *trlbench* starts up with a projects window as shown in Fig. 7.3. The listbox at the left side of the window shows the existing projects. Users

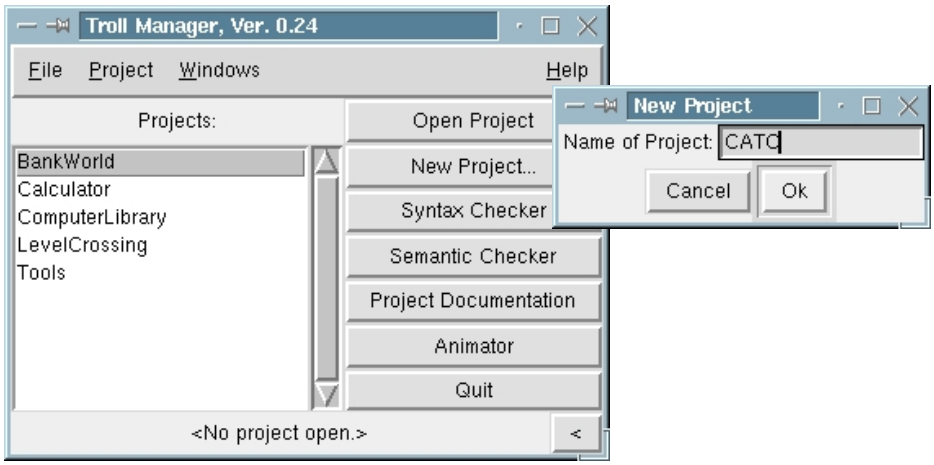


Figure 7.3: *trlbench* – Projects Window

can open an existing project or create a new one. For instance, in Fig. 7.3 a new project for the CATC example is created. A project consists of one or several files that together form a specification. For each project, an internal

file stores the names and location of the TROLL files as well as setup information that is necessary for the animation such as the database and C++ library names. By clicking on the buttons at the right side of the projects window, users can check the syntax and static semantics of the specification, call the documentation tool and animate the specification. Files belonging to a project are specified in a separate window. Fig 7.4 shows this window listing the files contained in the CATC project. Files do not necessarily need

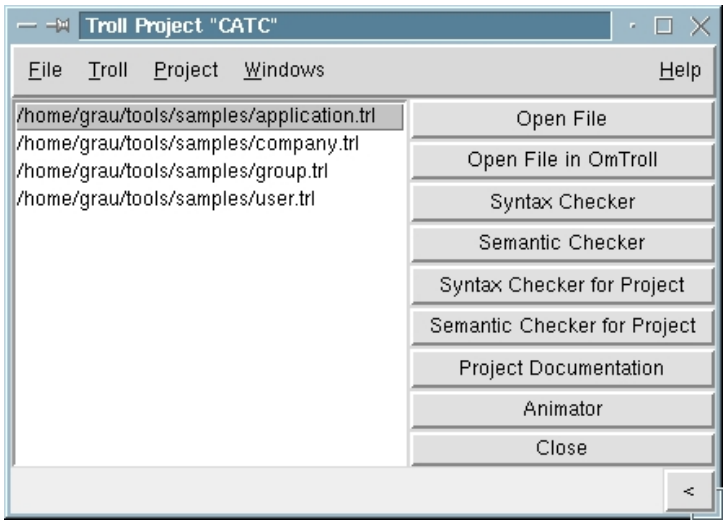


Figure 7.4: *trlbench* – Files Window

to be created in *trlbench*. Existing files, created in any ASCII editor or belonging to a different specification project, can just be added to the project files. This also allows to have a library of TROLL files that can be reused in several specification projects. The form in which the specification is structured through the project files is not determined by the tools. In this way, users have the freedom to structure the specification in files as they prefer. In the files window, users can select one or several files from the left listbox and then, by clicking on the respective buttons at the right side, open them in the editors or check their syntax and static semantics. As in the projects window, in the files window users may call the checkers, the documentation tool and the animator for the complete specification. When the documentation tool or the animator are called, *trlbench* may automatically determine which pieces of the building system need to be created or updated and calls

then the required tools. Fig. 7.5 shows the animation building window. In

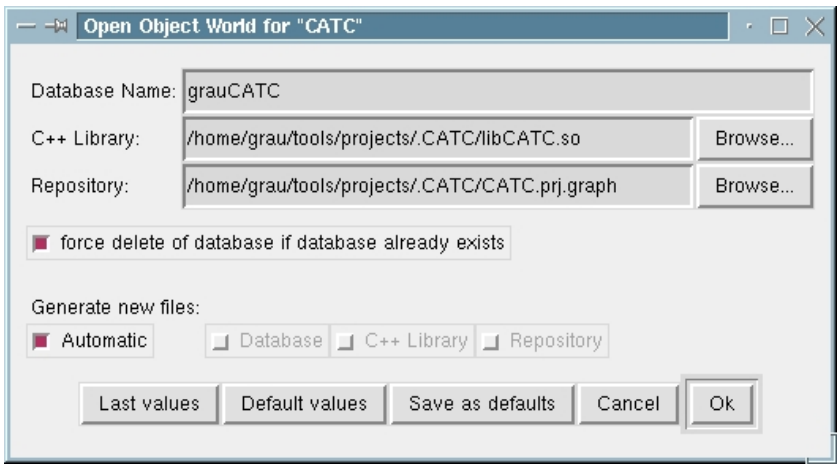


Figure 7.5: *trlbench* – Animation Building Window

this window, users may introduce the names of the instances database, the C++ library, and the repository, i. e. the file containing the syntax graph of the specification. Note that it is possible that users may want to have, for instance, several databases for the same specification in order to animate the specification with different objects populations. If the names are not introduced by the users, *trlbench* sets them to default names. By enabling/disabling the check buttons at the bottom of the window, users may define whether the database, the C++ library and the repository file have to be generated or just let *trlbench* automatically decide on it. In the latter case, *trlcheck* is only called to generate a new repository file if the file does not exist or is not updated with respect to the current TROLL files. Similarly, *trlcodgen* is only called to generate the C++ library if the library has not been created yet or corresponds to an older version of the specification. In the latter case, *trlcodgen* will compile only the new generated C++ code. Since the schema of the instances database is generated only from the static structure of the specification, changes in the specification do not necessarily mean the generation of a new database. This is important because the workbench does not support data migration and therefore the generation of a new database entails the loss of the objects already contained in the database. So whenever a database must be generated, *trlbench* calls *trldbgen* with an

option flag that allows to create an SQL script containing the commands used in the generation of the database schema. When the animator is called and the SQL script does not exist, *trlbench* calls *trldbgen* that creates the database schema. If the SQL script exists but it is not updated with respect to the TROLL files, *trlbench* calls *trldbgen* with an option that generates the SQL script but does not create the database. *trlbench* compares then the new SQL script with the old one. If the scripts are distinct, then *trlbench* calls *trldbgen* again which this time generates the database.

7.2.2 *trlgred* - TROLL Graphical Editor

trlgred is a graphical editor that allows developers to give a first overview of the model using OMTROLL diagrams. The editor supports the following diagrams:

- Community and object class declaration diagrams
- Communication diagram
- Data type diagram

As usual in class diagrams, object class declaration diagrams are included in the community diagram. The current version of *trlgred* does not support the modelling of object behaviour diagrams. Fig. 7.6 shows the community diagram editor of *trlgred* with the CATC example. As the OMTROLL diagrams of the CATC system were already presented in Chapter 3, we do not describe them here in further detail and concentrate on the functionalities of the editor. By selecting the corresponding check buttons at the top of the editor window, users can create object classes and define component and specialisation relationships between them. By clicking with the right-most button on a class, a class declaration window pops up. Here, users can introduce the object or component identification mechanism and the class signature, i. e. the declaration of attributes and actions for the class. For each attribute, the name, the type and special properties may be defined. Analogously, for each action, the name, input/output parameters with their respective types and special features may be introduced. The editor has a number of built-in checks to maintain the consistency of the data, allowing only syntactically correct constructs to be entered. In the community diagram, a double-click on a class shows/hides the signature of the class. The rest of diagram editors

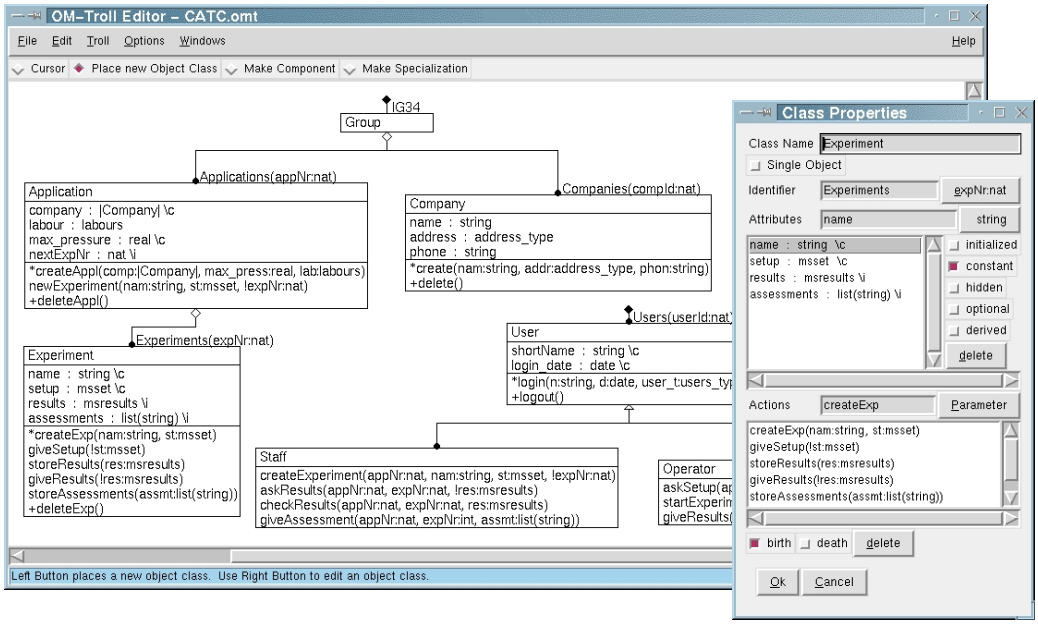


Figure 7.6: *trlgred* – Community Diagram

are depicted in Fig. 7.7. User-defined data types are specified in the data type diagram editor. Users may choose a type constructor by clicking on the check buttons at the top of the window. A dialog box then pops up and users may enter the data type name and the respective parameters. Data types specified in the data type diagram are automatically available in the community diagram to allow their use in the declaration of attributes and action parameters. The communication structure between objects classes can be specified in the communication diagram editor. In this editor, users may select classes defined in the community diagram and draw communication relationships between their actions.

OmTroll diagrams are automatically translated into Troll. The generated code includes the structural and communication parts of the specification. The remaining parts of the specification must then be completed using a text editor. The generation of Troll from the OmTroll diagrams available in *trlgred* is straightforward. The data type diagram is directly mapped into data type definitions. Troll classes and their signatures are

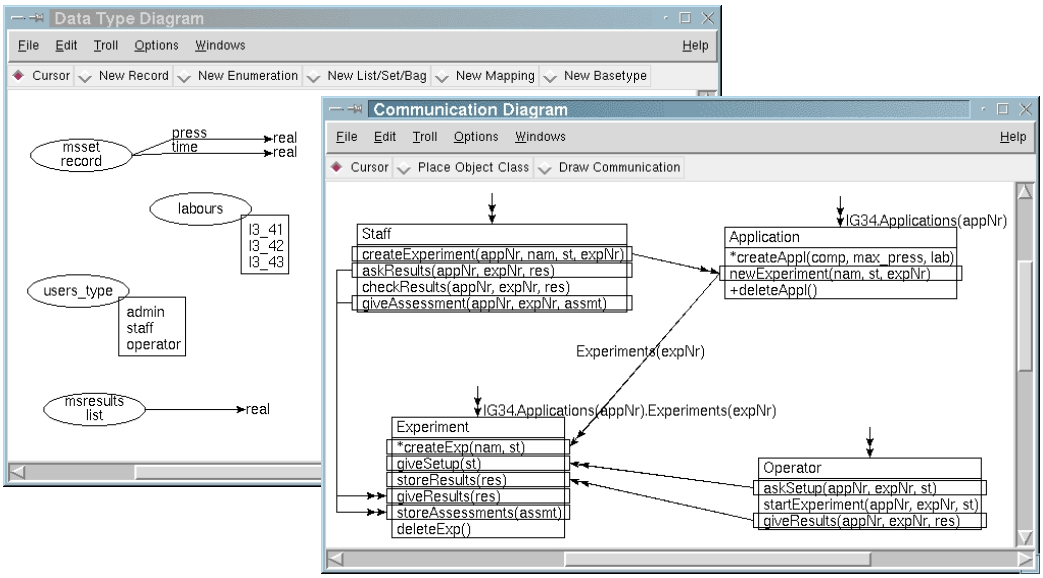


Figure 7.7: *trlgred* – Data Type and Communication Diagrams

obtained from the community diagram. Additionally, initialisation values for *initialised* attributes as well as derivation terms for *derived* attributes must be introduced. Object, component and specialisation declarations are also obtained from the community diagram. Specialisation conditions must be specified textually. Communication relationships defined in the communication diagram are mapped into action calling statements in the respective action behaviour definitions. The rest of the behaviour specification, i. e. the effect of actions on the attributes, assignments to output parameters and the specification of integrity constraints must be completed in TROLL. A reverse generation of OMTROLL diagrams from TROLL specifications is also possible. In this way, users may have a graphical representation for a subset of the TROLL specification in any modelling stage.

Currently, we are working on extending *trlgred* to allow users the introduction of the complete specification. Parts of the specification which have no correspondence in the OMTROLL diagrams would be entered in text windows and directly noted in TROLL syntax. In this case, no additional text editor would be necessary.

Another possible extension of *trlgred* is the inclusion of a new editor that supports the modelling of object behaviour diagrams. In this case, the generation of TROLL code presents the difficulty that behaviour diagrams, in contrast to all other OMTROLL diagrams, cannot be directly translated into TROLL. Behaviour diagrams express knowledge about the sequencing of actions (an action may only occur after another action has occurred). This is usually represented in the textual specification by action preconditions. Apart from the initial and final states, the rest of states in the behaviour diagram are identified by a name, possibly such that the name gives some intuition about the meaning of the state. A possible automatic translation would consist in generating in the TROLL class an attribute that explicitly represents the state of the object as given in the behaviour diagram. The data type of the attribute would be an enumeration whose labels are the names of the possible states. For each action that appears in the behaviour diagram, a precondition is then generated. The precondition guarantees that the action may only occur, if the current object's state, represented by the value of the generated attribute, corresponds to a state in the behaviour diagram from which the action causes a transition. Additionally, the assignment of the new state value to the attribute representing the state is generated in the behaviour definition part of the action. Although this solution would assure a correct sequencing in the occurrence of actions as indicated in the behaviour diagram, it could lead to an overspecification. A state of the behaviour diagram is usually represented by values of one or several attributes specified in the class. So the satisfaction of action preconditions should directly depend on these attributes and not on a special attribute representing explicitly the state of the object. A better solution would be to let the user decide if the generation of code should be carried out. The generated code could then be used as a hint and should be substituted with the correct code by the user.

Details about the implementation of *trlgred* can be found in [Har97b].

7.2.3 *trlted* - TROLL Textual Editor

trlted is a TROLL language mode for the (X)Emacs editors[4]. The language mode customises and extends these editors for supporting the editing of TROLL files. Another possibility we considered for the construction of TROLL

[4]Both editors are freely available. Information about these editors can be found at http://www.xemacs.org and http://www.gnu.org/software/emacs/emacs.html

specifications was the development of a TROLL *syntax-directed* editor. Such an editor would assist developers not familiar with the TROLL syntax in the construction of their first models and also obviate the need for a separate syntax check. Nevertheless, we thought the editor would be more useful if it were free-form. In this way, users have more freedom in the modelling process and may structure the specification as they prefer. Fig. 7.8 shows a screen dump of XEmacs in TROLL mode. The editor mode changes automat-

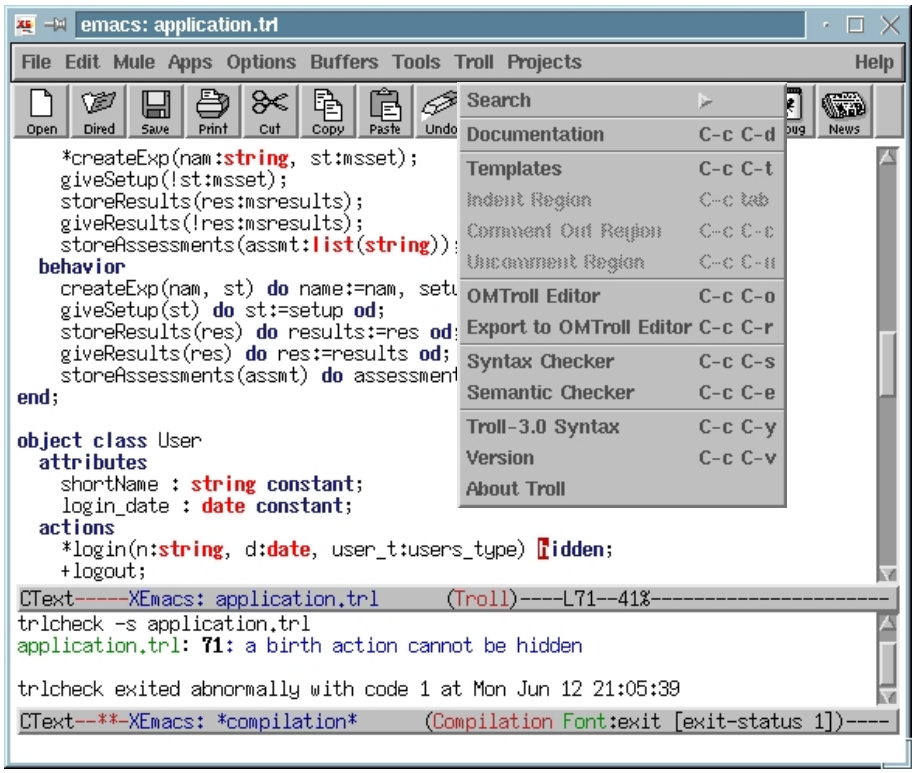


Figure 7.8: *trlted* – TROLL Language Mode in the XEmacs Editor

ically to TROLL mode when a file which extension "trl" is visited. Different colours and font styles highlight keywords, constants and comments. Functions added to the editor are accessible through menu entries and accelerator keys. Among these functions are:

- Searching for specification components through the project files.

- Generation of documentation files. These files contain structured information about elements declared in the specification (data types, classes, etc.) together with the file name and the line number in which they are declared.
- Quick insertion of specification templates. For instance, by a combination of keys the skeleton of a TROLL class is automatically inserted in the cursor position.
- Automatic indentation.
- Invocation of other TROLL tools (checkers, graphical editor, etc.).
- On-line information about TROLL (syntax, method, etc.).

As illustrated in Fig 7.8, the syntax and static semantics checkers are embedded in the editor. If some errors are detected, the XEmacs window will be split into two, showing in a checker window the list of errors. By clicking on an error line in the checker window, the file containing the error is visited and the cursor is directly positioned on the line where the error occurred.

Details about the Lisp implementation of a previous version of the TROLL mode can be found in [Sch97].

7.2.4 *trlcheck* - TROLL Syntax and Semantic Checker

trlcheck verifies if the syntax and static semantics of the specification are correct. During the analysis, *trlcheck* generates an abstract syntax graph from the specification which is used by the other TROLL tools. The analysis phases and the structure of the syntax graph have been presented in Chapter 4. *trlcheck* can be called from *trlbench*[5], the (X)Emacs editors or directly from the command line. The specification may be contained in one or several ASCII files. The syntax and semantic checkers can be called separately. So users may assure the syntax correctness of the specification (or of some parts of the specification) without additionally doing a semantic analysis. If *trlcheck* is called to do only a syntax analysis and if no syntax errors are found, the generated parse tree (or syntax graph) is stored in a file. The semantic analysis is done if *trlcheck* is called with the "-s" option and one or

[5]If the text editor selected in *trlbench* is either Emacs or XEmacs, the output from *trlcheck* is directly shown in the editor. If not, the errors are reported to a log window.

several file arguments. The file(s) may contain either the ASCII specification or the parse tree of the specification. In the first case, a syntax analysis is done before the semantic analysis. In the second case, if the specification was parsed in several parts, the parse trees are first merged and then the semantic analysis is directly done. So calling the semantic checker does not require to do the syntax analysis if it was done before. After the semantic analysis and if no errors are found the resulting abstract syntax graph is stored in a file.

The scanner and the parser are grouped into a single pass and are implemented with the help of the compiler generators `Flex` and `Bison`[6]. `Flex` is a scanner generator. This tool reads a specification of regular expressions describing the tokens of the TROLL language and generates a C program containing a scanner for TROLL specifications. The scanner consists of a transition table for a deterministic finite automaton (DFA) constructed from the regular expressions and a DFA simulator that uses the table to recognise lexemes in the TROLL specification. `Bison` is an LALR(1) context-free grammar parser generator. This tool reads a TROLL context-free grammar specification and generates a C program to parse TROLL specifications. For each rule in the grammar specification, an action may be defined that will be executed when an instance of the rule is recognised during the parsing. These actions will construct the parse tree using the graph library presented in Sect. 4.2. The parser consists of a finite-state stack machine that is represented by a set of parsing tables, a stack and a driver function. Each state corresponds to a stage in the grammar rules and summarises the information contained in the stack below it. The combination of the state at the top of the stack and the look-ahead token are used to index the parsing tables and determine the shift-reduce parsing decision. The parser reads a sequence of tokens coming from the scanner and groups them using the grammar rules. As it does this, the actions associated to the grammar rules are executed, i. e. the corresponding subtrees in the parse tree are created. If the specification is syntactically correct, the entire token sequence is reduced to a single grouping whose symbol is the grammar's start symbol.

In case of syntax errors, it is usually not acceptable to terminate the parsing after detecting the first error because subsequent processing of the specification may discover additional errors. There are several strategies that

[6] `Flex` and `Bison` are free software versions of the well-known tools `Lex` and `Yacc` respectively. Information about `Flex` and `Bison` may be found at their respective home pages http://www.gnu.org/software/bison/bison.html and http://www.gnu.org/software/flex/flex.html.

a parser can employ to recover from a syntactic error [ASU86]. In the parser generated by `Bison`, error recovery can be performed increasing the grammar with error productions. The parser generates a special nonterminal token *error* whenever a syntax error is detected. If there is a rule in the grammar that recognises this token in the current context, the syntax error can be reported and the parser can continue. The error productions are positioned in the grammar in typical places of errors. Common punctuation errors are, for instance, the omission of a semicolon at the end of a line and the use of a semicolon in place of a comma in the parameter list of an action declaration. A typical example of an operator error is to leave out the colon from the assignment operator (`:=`). Sometimes, an error recovery may cause several *false* errors that were not made by the user, but they were introduced by changes made to the parser state during error recovery. To prevent this, the parser requires that after the detection of an error, three consecutive tokens must be successfully parsed before permitting another error message. Since this is not always necessary, errors messages may be resumed immediately by using a special macro in the action specified in the error rule. Error messages should be understandable and report as much information as possible. When an error is detected, the error-handling routine in the `Bison` parser just reports a `"Syntax Error"` message. This routine may, however, be rewritten for giving more expressive error messages. In the TROLL parser, an error message includes the token where the error was detected, its localisation in the source code (file name and line number) and a list of tokens that were expected at the place of the error. For the latter and based on [SF91], the `Bison` parser had to be modified in order to obtain from the parsing tables and for a determined parser state which tokens lead to a valid state. For a concrete example, consider the next extract from the CATC specification with some syntax errors:

```
...
actions
  *createAppl(comp:|Company| max_press:real, lab:labors)
  newExperiment(nam:string, st:msset, !expNr:nat);
...
```

When parsing the specification, *trlcheck* reports the following error messages:

```
application.trl: 26: before "max_press" expected: ‘,’ , ‘)’
application.trl: 27: before "newExperiment" expected: ‘;’
```

The error messages produced by the syntax checker can sometimes be misleading. In some cases, a syntax error may have occurred long before the position at which its presence is detected, and the precise cause of the error may therefore be very difficult to deduce.

Once the syntax correctness of the specification has been checked, the next step is to check its static semantics. The semantic checker looks for semantic errors through the syntax graph and reports the errors to the specifiers. As it does this, the syntax graph is extended with new edges representing context-sensitive information. The semantic analysis and the list of semantic rules to be checked statically have already been presented in Sect. 4.3. Semantic errors include incorrect values applied to action calls, incompatible types in operations and assignments, use of undefined variables and violation of TROLL context-sensitive rules such as declaration of birth actions in specialisation classes. As the syntax checker, the semantic checker does not stop after detecting the first error and continues looking for additional errors. Errors messages are self-explanatory. Details about the implementation and the list of error messages that may be returned by the checker can be found in [Rüt99].

If the semantic checker does not detect any errors, the resulting syntax graph is stored in a file. The graph structure was presented in Fig. 4.2 on page 62. *trlcheck* has an option that allows, for debugging tasks, to show the graph in a text format. We show next an extract from the graph generated from the CATC specification:

```
V_systemSpec (8, 1, CATC)
  E_dataTypeSpec_list (13, 8 -> 9)
    V_dataTypeSpec_list (9, 1, no info)
      ...

  E_objectClassSpec_list (314, 8 -> 194)
    V_objectClassSpec_list (194, 1, no info)
      E_first (315, 194 -> 193)
      E_elem (317, 194 -> 193)
        V_objectClassSpec (193, 10, application.trl)
          E_class_id (308, 193 -> 67)
            V_class_id (67, 10, Application)
      ...

  E_objectDecl_list (514, 8 -> 320)
```

```
    V_objectDecl_list (320, 1, no info)
      ...

  E_behaviorSpec_list (786, 8 -> 464)
    V_behaviorSpec_list (464, 1, no info)
      ...
```

The arguments in a vertex show its identification value, the line number in the text file in which the token appears and a value whose meaning depends on the vertex type. The arguments in an edge represent its identification value and the initial and terminal vertexes. The root vertex, `V_systemSpec`, has four children vertexes containing the data type, object class, object and global behaviour definitions of the specification. The first element in the list of object class specifications is the class `Application` that is specified in the file `application.trl`, line number `10`.

The remaining TROLL tools access the graph by a common interface. So changes in the graph structure do not require changes in these tools. The graph interface provides access functions at an abstract level. These functions return, for instance, the attributes for a given class or the operations defining the behaviour of a given action. The graph interface and its implementation are described in [Voe99].

7.2.5 *trldoc* - TROLL Documentation Tool

trldoc reads the graph generated from the specification by *trlcheck* and generates HTML code. The generated code can be viewed in any HTML-browser supporting Java Scripts and frames. The code allows hypertext navigation through the specification and the introduction of informal comments to document the model. Additionally, *trldoc* allows to print the complete specification together with the informal comments in one document. *trldoc* can be considered as an unparser that adds the missing concrete syntax of the specification to the abstract syntax contained in the graph to obtain a structured representation of the original text specification.

Fig. 7.9 shows a screenshot of Netscape browsing the HTML code generated from the CATC specification. The window's layout consists of four frames. The left frame contains links to the main declaration parts of the specification: data types, object classes, objects, and system behaviour. Clicking on one of these links shows the corresponding declaration list on

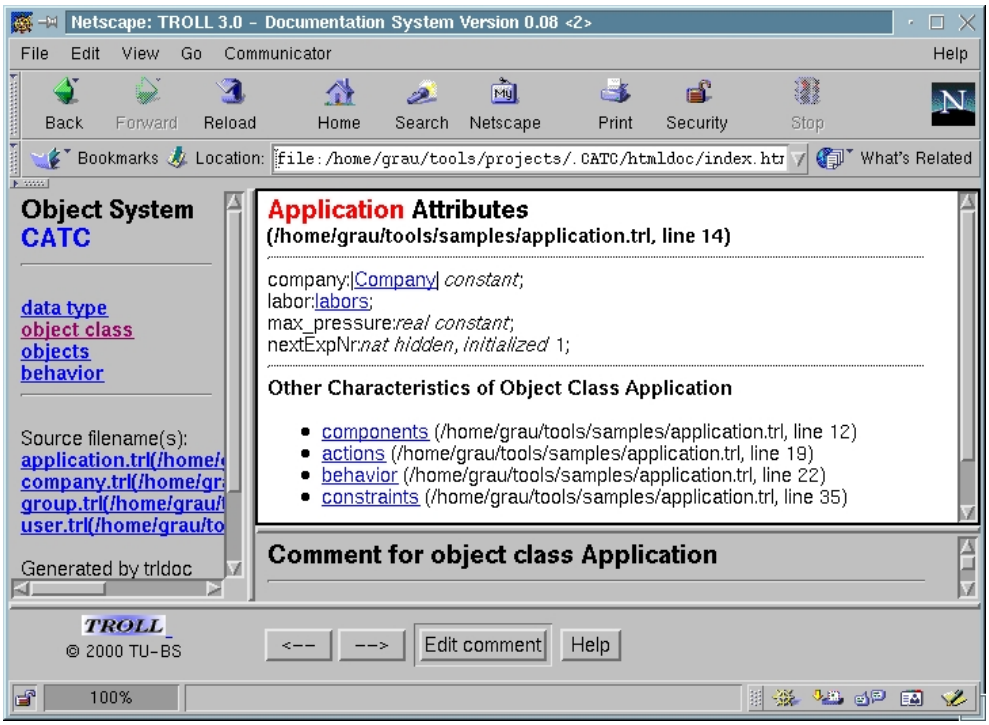


Figure 7.9: *trldoc* – Hypertext Navigation through the Specification

the main frame at the right side of the window. Under object classes, a list of all classes defined in the object system is displayed together with the source file names and line numbers in which the classes are defined. If an object class is selected from this list, links to all the parts specified within the class (specialisations, components, attributes, actions, local behaviour definitions and constraints) are displayed. Following these links, the corresponding definitions are shown. Definitions provide further links to other parts of the specification (component classes, specialisations, data types, etc.). The TROLL source code files may be displayed on the main frame by clicking on the corresponding links at the bottom of the left frame. Comments for the current displayed declaration are shown on the frame below the main frame. Pressing the buttons with the back/forward arrows on the bottom frame redisplay the documents that were loaded before/after the current one. The help button gives instructions on using *trldoc*. The TROLL logo at the left side displays information about the TROLL project. Informal comments

for each specification component can be introduced by pressing the "Edit comment" button. A textarea entry then appears at the comments frame. This is depicted in Fig. 7.10. Comments are directly edited in HTML. So

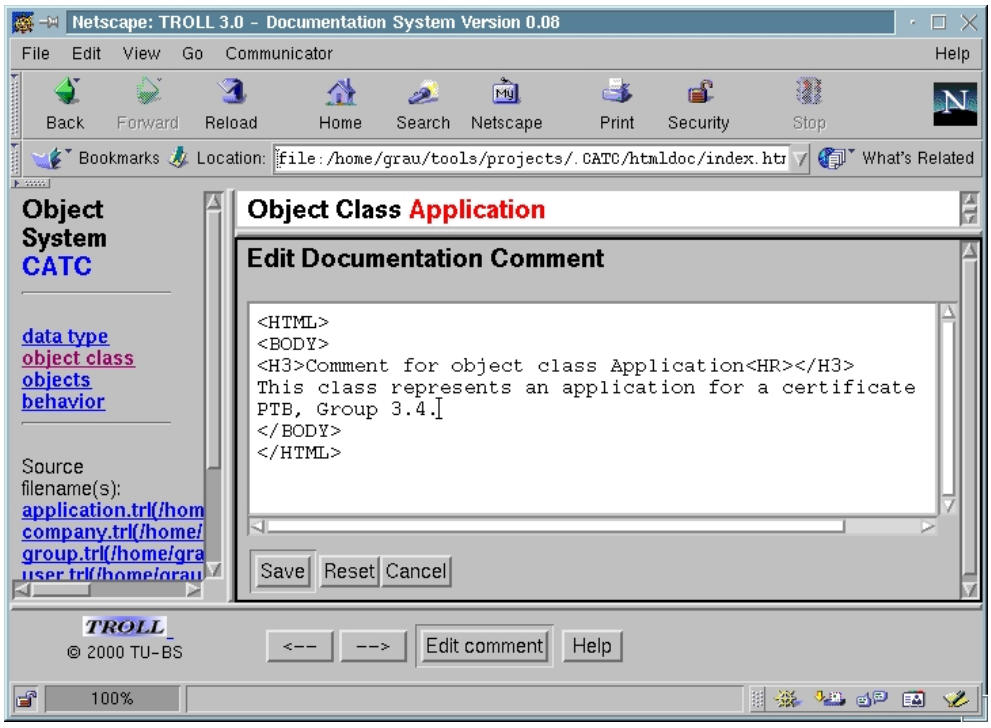


Figure 7.10: *trldoc* – Introduction of Informal Comments

it is possible to use different fonts and colours and to introduce not only text but also figures, etc. Below the textarea, there are buttons for saving the comment, restoring the original comment before the current changes or cancelling without saving.

trldoc can be called with an option to generate a Postscript file containing the TROLL specification together with the informal comments. To do this, *trldoc* uses the HTML-Postscript generator `html2ps`[7]. *trldoc* has been implemented in C++ and Perl. For implementation details, the reader is referred to [Ger00].

[7] `html2ps` is a free software tool. Information about this tool may be found at its home page http://www.tdb.uu.se/~jan/html2ps.html.

7.2.6 *trldbgen* - TROLL Database Generator

trldbgen reads the static structure of the specification contained in the syntax graph and generates a database schema. The database stores the object instances created during the animation. *trldbgen* has been implemented in C++ and uses Postgres as DBMS. In order to generate the database schema, *trldbgen* makes a connection to the Postgres backend server and sends then the corresponding SQL commands. As mentioned in the presentation of *trlbench*, this tool can also be called with an option that stores the generated SQL statements into a file to be processed later. The database generation is based on the transformation rules presented in Chapter 5. In the next, we show some of the SQL commands used in the database definition of the CATC system.

The generation of surrogate keys (SOIDs) for the objects contained in the database is implemented using the sequence constructor provided by Postgres. A sequence number generator is created as follows:

```
CREATE SEQUENCE OID_seq increment 1 start 1
```

After the sequence is created, the function `nextval(OID_seq)` returns a new value from the sequence.

During the animation, the persistence layer uses some special tables to find out the objects in the database. These tables are created as follows:

```
CREATE TABLE hierarchy (
  complex_table text, child_table text, type text NOT NULL,
  PRIMARY KEY (complex_table, child_table))
CREATE TABLE classes (
  table_name text, class_name text NOT NULL,
  PRIMARY KEY (table_name))
CREATE TABLE complex_attributes (
  complex_table text, child_table text, type text NOT NULL,
  PRIMARY KEY (complex_table, child_table))
```

The table `hierarchy` stores the hierarchy relationships between tables. The type of relationship is stored in the column `type` (either "component" or "specialisation"). For each table in the database storing TROLL objects, the table `classes` stores the name of the corresponding TROLL class. The table `complex_attributes` stores the relationships between tables representing TROLL objects and those representing complex attributes. The column `type` stores the constructor type of the attribute (list, set, etc.). These tables are

instantiated by the database generator during the creation of the respective tables. For example, let _ig34 and _applications_1 be the tables storing the objects IG34 and applications respectively, the following commands are generated:

```
INSERT INTO classes VALUES ('_ig34', 'Group')
INSERT INTO classes VALUES ('_applications_1', 'Application')
INSERT INTO hierarchy VALUES ('_ig34', '_applications_1', 'comp')
```

The table _ig34 contains objects of class Group and the table _applications_1 contains objects of class Application that are components of the objects contained in the table _ig34.

Postgres does not support the definition of foreign keys. So referential integrity constraints are implemented by triggers. For instance, the table _applications_1 and the component relationship between applications and IG34 are defined as follows:

```
CREATE TABLE _applications_1 (
  soid int DEFAULT nextval('OID_seq'),
  complex int NOT NULL, par int NOT NULL CHECK (par >= 0),
  PRIMARY KEY (soid), UNIQUE (complex,par))
CREATE FUNCTION check_cascade_applications_1_ig34 ()
  RETURNS opaque AS 'BEGIN DELETE FROM _applications_1;
                     WHERE OLD.soid = complex;
                     RETURN OLD; END;' LANGUAGE 'plpgsql';
CREATE TRIGGER tg_check_cascade_applications_1_ig34
  BEFORE DELETE ON _ig34 FOR EACH ROW EXECUTE
  PROCEDURE check_cascade_applications_1_ig34 ()
```

As defined in the transformation rule 5.2.2 on page 95, the table containing the component objects includes a column (complex) that refers to the primary key of the table containing the compound objects. The trigger defines that whenever an object of the compound table (_ig34) is deleted, then all its components are deleted from the component table (_applications_1). Note that since the SOID of an object never changes, referential integrity checks in *update* operations are not required. On the other hand, the referential integrity in *insert* operations in the component table is assured directly by the persistence layer.

Once the table _applications_1 has been created, the attributes of the class Application are mapped into columns of this table as follows:

```
ALTER TABLE _applications_1 ADD (_labour text NOT NULL
  CHECK (_labour in "l3_41", "l3_42", "l3_43"))
ALTER TABLE _applications_1 ADD (_max_pressure real NOT NULL)
ALTER TABLE _applications_1 ADD (_nextexpnr int NOT NULL
  CHECK (_nextExpNr >= 0))
ALTER TABLE _applications_1 ADD (_company int NOT NULL)
CREATE FUNCTION check_ref_applications_1_companies_1 ()
  RETURNS opaque AS 'DECLARE rec RECORD;
                     BEGIN SELECT * INTO rec FROM _applications_1
                     WHERE OLD.companies_1_soid = _company;
                     IF FOUND THEN RETURN NULL; ELSE RETURN OLD;
                     END IF; END;' LANGUAGE 'plpgsql';
CREATE TRIGGER tg_check_ref_applications_1_companies_1
  BEFORE DELETE ON _companies_1 FOR EACH ROW EXECUTE
  PROCEDURE check_ref_applications_1_companies_1 ()
```

Attributes declared as non-optional have a `NOT NULL` constraint in the respective column. Enumerations (e. g. `_labour`) are implemented as strings and an integrity constraint that limits their values to the label names defined in the enumerations. Natural numbers (e. g. `_nextexpnr`) are represented by integers that must be greater or equal to zero. An object-valued attribute (e. g. `_company`) contains the SOID of the referenced object. In the presentation of the mapping rules (see rule 5.2.5 on page 102) and in order to assure the referential integrity, this attribute was defined as a foreign key to the primary key of a special table including the SOIDs of all existing objects of the referenced class. This table was necessary because objects of the same class may be stored in different tables and only one table may be referenced in the definition of a foreign key. Since referential integrity constraints are implemented by triggers, these tables are not necessary. In the example above, a trigger assures that a company (i. e. an object contained in the table `companies_1`) may be deleted only if it is not referenced by any application (i. e. the SOID of the company object is not contained in the column `_company` of any tuple in the table `_applications_1`). If a reference to the company is found in an application object, the trigger returns a `NULL` value and the animator then reports to the user that the company may not be deleted. In case there would be more than one table containing objects of class `company`, a similar trigger would be defined for each of these tables. The persistence layer uses the information contained in the table `classes` for finding out the table in which the referenced object is stored. As in components, referential integrity checks

in *update* operations on the referenced table are not necessary because the SOID of an object never changes. Referential integrity in *update* and *insert* operations on the table containing the reference attribute is assured by the persistence layer.

For further details about the implementation of *trldbgen* and the database interface see [Voe99, Sch99].

7.2.7 *trlcodgen* - TROLL-C++ Code Generator

trlcodgen is a C++ code generator of TROLL specifications. *trlcodgen* reads the specification contained in the syntax graph and generates C++ code that implements the behaviour of the objects. Additionally, the C++ code includes dynamic checks, access to the database interface to retrieve/update the state of the objects and code to follow the execution trace. The code generator itself has been implemented in C++. The transformation of a TROLL specification into C++ code was already explained in Chapter 6. In this section, we show some of the code generated from the CATC specification.

A C++ class is generated for each TROLL class. The code corresponding to each C++ class is divided into a header and an implementation file, with the name of the class suffixed `'_.h'` and `'_.cc'` respectively. The class is defined in the header file and the implementation of the member functions is placed in the implementation file. An additional implementation file, `execute.cc`, is generated containing the code which is in scope for all generated classes (`call_extensions, call_global_interactions`, etc.).

The signature of the TROLL class is defined in the corresponding C++ header file. For instance, the signature of the class `Application` (see example 3.3.2 on page 47) is mapped into the C++ class definition as follows:

```
class Application : public troll_class {
 public:
  Application(const identList& ident) : troll_class (ident){};
  // attributes
  oid<Company> company_;
  oid<Company> company_new;
  tstring labour_;
  tstring labour_new;
  treal max_pressure_;
  treal max_pressure_new;
  tnat nextexpnr_;
```

```
  tnat nextexpnr_new;
  // components
  tmap<tnat,Experiment*> experiments_;
  // methods
  void createappl_(const oid<Company>& comp_,
                   const treal& max_press_,
                   const tstring& lab_);
  void newexperiment_(const tstring& nam_,
                      const msset_rec& st_,
                      tnat& expnr_);
  void deleteappl_();

  void check_constraints(const bool& initial=false);
  oid<Company> get_company_new();
  tstring get_labour_new();
  treal get_max_pressure_new();
  tnat get_nextexpnr_new();
  void init_attributes();
  void read_attributes();
  void write_attributes();
}; // class Application
```

The C++ class `Application` is a derived class of the class `troll_class`. The constructor calls the constructor of the superclass with the object identifier as argument (`ident`). For each attribute defined in the TROLL class, there are two attributes: one containing the value in the current state (e. g. `company_`) and another one containing the value in the new state (e. g. `company_new`). The multiple component `Experiments` is represented by a map with domain the parameter identifiers and range pointers to objects of class `Experiment`. Actions (`createAppl`, `newExperiment` and `deleteAppl`) are directly mapped into function members in the C++ class. Parameters are passed by reference. Input parameters are declared *const*. For each attribute in the TROLL class, a function is defined that returns the value of the attribute in the new state (e. g. `get_company_new`). The rest of functions initialise attributes, check integrity constraints and read and write attributes values from/to the database.

The implementation of the member functions is placed in the implementation file. For instance, the behaviour definition of the action `newExperiment` (see example 3.3.4 on page 49) represented by the function `newexperiment_` is implemented as follows:

```
void Application::newexperiment_(const tstring& nam_,
                                 const msset_rec& st_,
                                 tnat& expnr_) {
  // Control Code
  vector<local_trace>::iterator i;
  for (i = trace.begin(); i != trace.end(); i++)
    if (i->action == "newExperiment")
      if (equal(i->params[0],nam_) && equal(i->params[1],st_)){
        assign_param(i->params[2],expnr_);
        return;
      }
  int pos = to_trace("newExperiment",&nam_,&st_,&expnr_,NULL);

  // Behaviour Code
  if (!(st_.press_ <= max_pressure_))
    throw_exception("Precondition not fulfilled:
                     onlyIf(st.press<=max_pressure)");
  if (verbosity)
    show_in_console("calling Experiments(nextExpNr).
                     createExp(nam,st)");
  get_comp(experiments_,nextexpnr_,"Experiment","Experiments",
           "tnat",BIRTH);
  experiments_[nextexpnr_]->createexp_(nam_,st_);
  if (verbosity)
    show_in_console("assigning expNr := nextExpNr");
  expnr_ = nextexpnr_;
  if (verbosity)
    show_in_console("assigning nextExpNr := nextExpNr+1");
  nextexpnr_new = nextexpnr_+1;

  //  Control Code
  to_params(pos,nam_); to_params(pos,st_); to_params(pos,expnr_);
  call_extensions("newExperiment",&nam_,&st_,&expnr_,NULL);
  call_global_interactions("newExperiment",&nam_,&st_,&expnr_,NULL);
}
```

The code in the *for* loop checks if the action `newExperiment` with the same input parameter values is contained in the local trace, i. e. if the action was already executed. If this is the case, the ouput parameter `expnr_` is assigned (function `assign_param`) and the function returns. If the action was not ex-

ecuted before, it is introduced in the trace by the function `to_trace`. Since the function may have any number of arguments depending on the parameters defined in the TROLL action, a `NULL` value at the end of the argument list allows the function to know that all arguments were read. Additionally, this function checks the termination conditions as presented in Chapter 6. Next, the action precondition is checked. If the precondition is not satisfied, the function `throw_exception` throws an exception and the whole state transition is rejected. If the user of the animator has set the animation to verbose mode on (the variable `verbosity` is true), the execution trace is shown in the animator console (function `show_in_console`). The function `get_comp` creates an object of class `Experiment_` with local parameter identifier `nextexpnr_` on main memory and assigns a pointer to this object to the component `experiments_[nextexpnr_]`. The action `createexp_` is called in this object. The attribute `nextexpnr_` is then assigned to the output parameter `expnr_`. The increment of the TROLL attribute `nextExpNr` in the next state is denoted by the assignment of `nextexpnr_+1` to `nextexpnr_new`. After the behaviour code has been executed, the parameter values are stored in the local trace and the functions `call_extensions` and `call_global_interactions` are called. These functions are defined in the superclass and call the corresponding global functions.

The next example shows an extract of the function `call_extensions` managing the specialisation of objects of class `User` in `Staff` (see example 3.3.3 on page 48):

```
void call_extensions(const string& class_name, identList& id,
                     const string& action, vector<void*> params) {
 if (class_name == "User")  {
    User* objUser;
    ex_man->get_object (id, &objUser);
    if (action == "login") {
      tstring n_ = *(tstring*) params[0];
      tdate d_ = *(tdate*) params[1];
      tstring t_ = *(tstring*) params[2];
      if (verbosity)
        show_in_console("checking specialisation condition of
                         Staff: t = \"staff\"");
      if (t_ == "staff") {
        if (verbosity)
          show_in_console("creating specialisation aspect Staff");
```

```
        Staff* objStaff;
        objUser->get_spec(objStaff,"Staff",BIRTH);
        objStaff->get_superclass(objStaff->superclass);}
...
```

The function arguments are the class name, the object identifier, and the name and parameter values of the action called in the object. Since the number and data type of the parameters depend on the action, they are passed in a vector of pointers to any type (`void*`). They must be converted (*cast*) to the corresponding type before they are used. If the object class is `User` and the action is `login`, the specialisation condition in `Staff` is evaluated. Since the satisfaction of the condition may depend on the parameter values, they are cast to the corresponding data types and assigned to local variables. The variable names are those defined in the specialisation declaration of the class. If the condition holds, the specialisation object is created (`objUser->get_spec`) and a pointer to the superclass is assigned to the attribute `superclass` of the specialisation. The C++ code generated from the CATC system is further presented in Appendix D.

By calling *trlcodgen* with an extra option, the generated code is compiled and linked in a shared library. For this, a `Makefile` is generated that contains all compilation parameters and relationships among the generated files. *trlcodgen* calls the `GNU make` utility to process this file. In order to save compilation time, if the specification has changed and *trlcodgen* is called again, the C++ code is first generated on main memory and then compared with the files containing the code generated for the previous version. If they are the same, the files are not overwritten. So only modified files are compiled again.

7.2.8 *trlanim* - TROLL Animator

trlanim is an animator of TROLL specifications. This tool allows users to simulate the occurrence of events in the system in order to observe the dynamics of the specification. In *trlanim*, users may create objects, navigate through their interfaces and select initial actions to be executed. During the animation, *trlanim* makes uses of the syntax graph generated by *trlcheck* to obtain information about the structure of the specification, the database generated by *trldbgen* to retrieve/update the state of the objects and the C++ library generated by *trlcodgen* to execute the behaviour associated to the objects'

actions. The animator incorporates an execution manager that monitors the execution of state transitions in the system according to the execution model presented in the previous chapter. Animating the specification serves two main purposes. On the one hand, it helps developers to find runtime errors in the specification that could not be statically detected by *trlcheck*. On the other hand, it helps developers and end users of the application to validate the specification against user requirements. To this end, the specification is checked in several scenarios to see if it meets the expected behaviour. *trlanim* has an intuitive user-friendly graphical interface that encourages especially end users, who are not necessarily familiar with the specification formalisms, to actively participate in the validation process. In the next, we describe the functionalities of *trlanim* by example. To do this we show a possible animation session of the CATC specification. In the session, we want to validate the specification in a scenario in which a staff user sets up an experiment for an application.

trlanim starts up with an object instances window as illustrated in Fig. 7.11. In this window, users can select an existing object to view or create

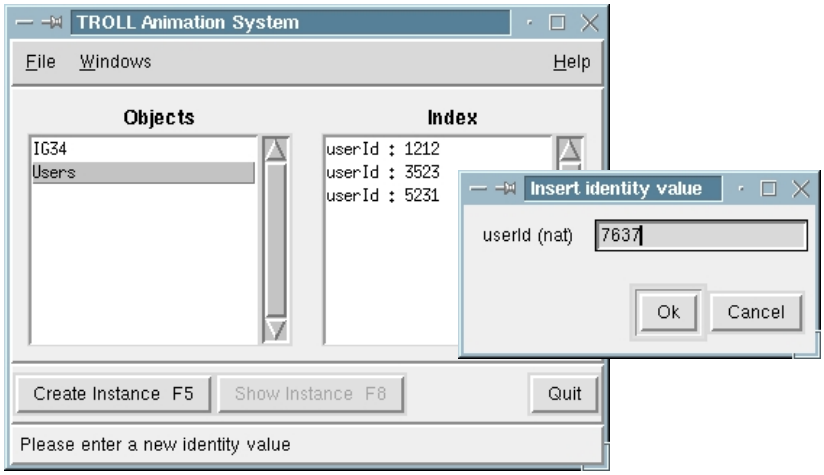


Figure 7.11: *trlanim* – Object Instances Window of CATC

a new one. The left listbox shows the names of the objects that can be or have been created in the system as declared in the object declaration part of the specification. In the CATC specification, these are `IG34` and `Users`. Clicking on an object's name belonging to a multiple object declaration shows the

parameter identifiers of existing objects in the right listbox. The interface of an object may be viewed by double-clicking on the object, or index in case of a parametrised object, or by pressing on the *Show Instance* button at the bottom of the window. To create an object, users must select the object's name in the left listbox and then press the *Create Instance* button. In case of a parametrised object, a window pops up and users must enter the parameter identifier. In order to validate the setup of an experiment by a staff user, a `Users` object must be created. As shown in the window at the right side of the figure, we are going to create an object `Users` with a parameter value `7637`. If the parameter value already identifies an object in the system, an error window reports of this and a new value must be introduced. An object is created by calling the birth action as declared in the respective class. If there exist several birth actions, their names are first shown in a window and users may select one of them. Following the example and after introducing the parameter identifier of an object `Users`, an action call window corresponding to the birth action (`login`) appears. This window is depicted at the left side of Fig. 7.12. In the action call window, values for the input parameters, if

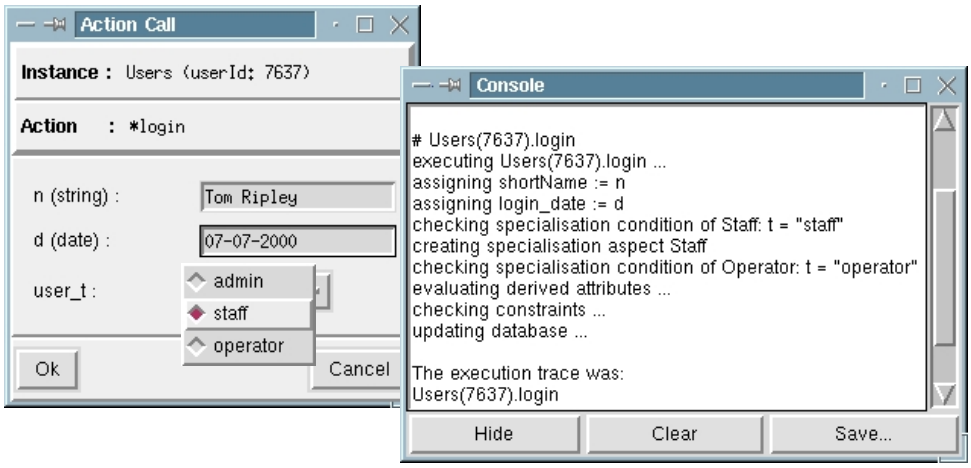


Figure 7.12: *trlanim* – Action Call Window of `login`

defined, are entered. The parameter names are those defined in the respective action declaration. In the example, we introduce the name of the user, the login date and the user type as `staff`. After pressing the *Ok* button the

action is executed. A console window, as illustrated at the right side of the figure, shows the execution trace. The parameters `n` and `d` are assigned to the attributes `shortName` and `login_date` respectively. Specialisation conditions are checked and the specialisation aspect `Staff` is created. In order to set up an experiment, we must first create an application. Applications are defined as components of the object `IG34`. Assuming we have previously created this object, we can select it in the object instances window (see Fig. 7.11). An instance view window, as illustrated in Fig. 7.13, pops up in which the object's interface is shown. An instance view window consists of four listboxes that

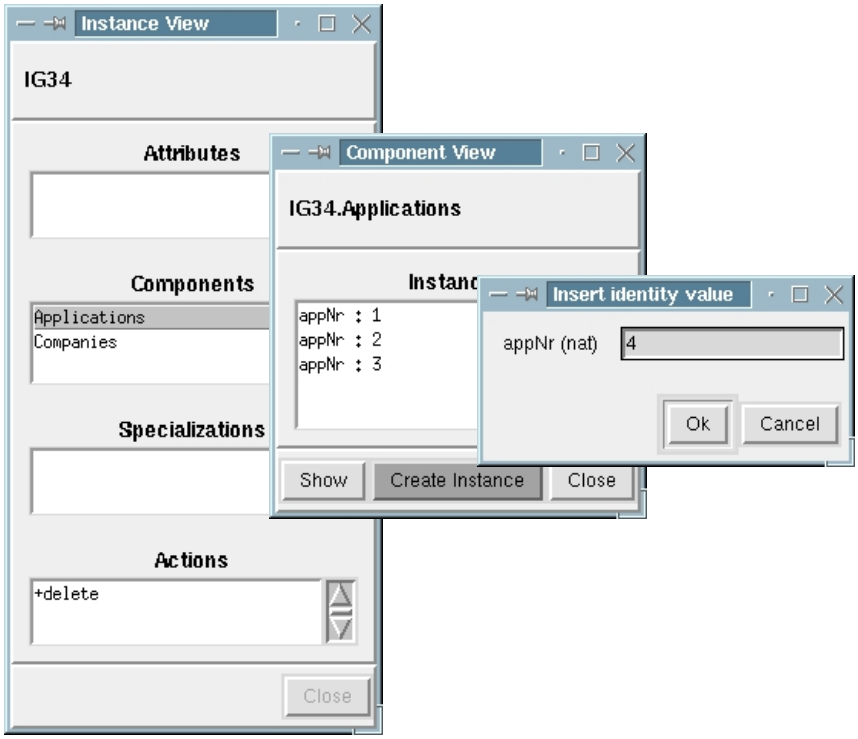


Figure 7.13: *trlanim* – Instance View Window of `IG34`

show, if declared, the attributes, components, specialisations and actions of the object. By double-clicking on a parametrised component, a component view window appears. Here, as in the object instances window, users can select a component object to view or create a new one. In the example, after selecting the component `Applications`, we are going to create an application

with parameter identifier `4`. Note that as objects are persistent, they can be used in several animation sessions and therefore participate in several scenarios. So in our example, we could have just used an existing application. We create a new one in order to show the functionalities of the animator. After introducing the parameter identifier of the application, an action call window corresponding to the birth action (`createAppl`) pops up, as displayed in Fig. 7.14. Here, values for the input parameters of the action are introduced.

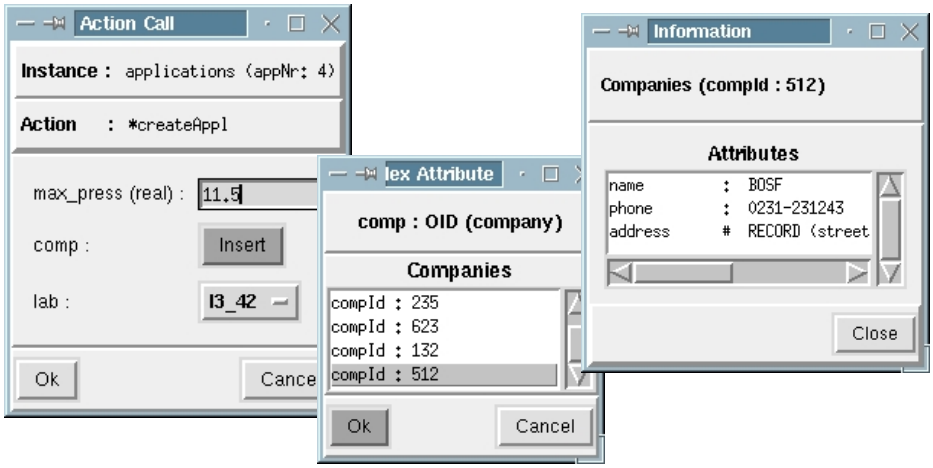


Figure 7.14: *trlanim* – Action Call Window of `createAppl`

The application is assigned to labour `l3_42`, and the maximum pressure that can be set up in an experiment for the application is set to `11.5`. The introduction of values representing references to objects is specially handled. A possible way of entering object references would consist in directly giving the identification path of the referenced object. Since identification paths may be very long, and it must also be assured that the referenced object exists, this solution would be error-prone. For this reason, the animator shows all existing objects of the referenced class in a list from which users must just select the referenced object. For instance, the data type of the input parameter `comp` is a reference to an object of class `company`. By clicking on the *Insert* button, a window lists the existing companies. To obtain more information about a company, a click with the right mouse button on the company shows the current values of its attributes. As mentioned earlier in the description

of the database generator, if the death action is called in a company which is referenced by an application, the animator reports of this to the user and the action does not take place. Once the application has been created, the staff user may create and set up experiments for the application. From the object instances window, we select the object `Users(7637)` that we created previously. Fig. 7.15 depicts the corresponding object's interface. The left

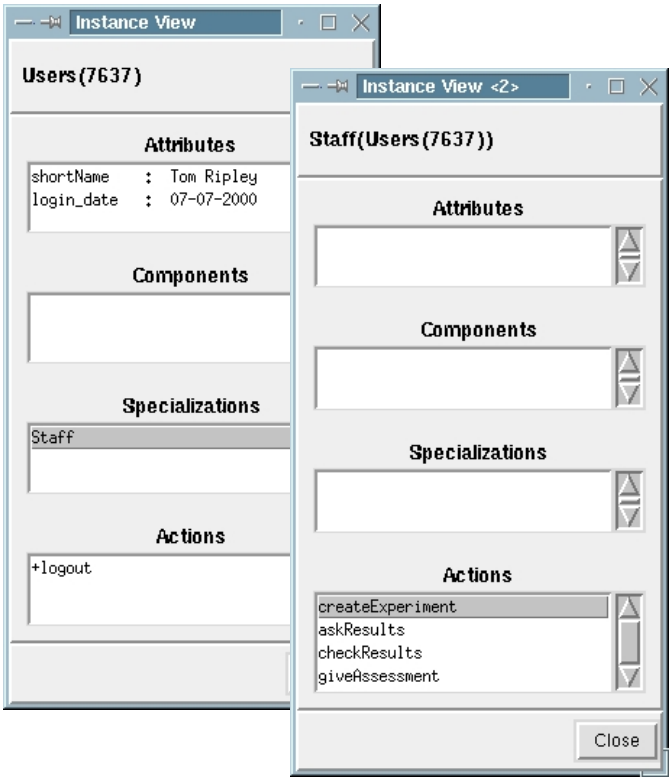


Figure 7.15: *trlanim* – Instance View Window of `Users`

instance view window shows the interface corresponding to the base class. Clicking on the specialisation aspect `Staff` leads to the right instance view window. The listbox at the bottom of the instance view window lists the actions that can be called in the object. These actions are those declared in the object class with the exception of birth and hidden actions. Birth actions cannot be called because the object already exists and hidden ac-

tions may only be called by local actions. If a death action is called and the object has components, a warning window informs the user that the action will also delete all the components. Clicking on the action `createExperiment` pops up the corresponding action call window, as displayed in Fig. 7.16. A

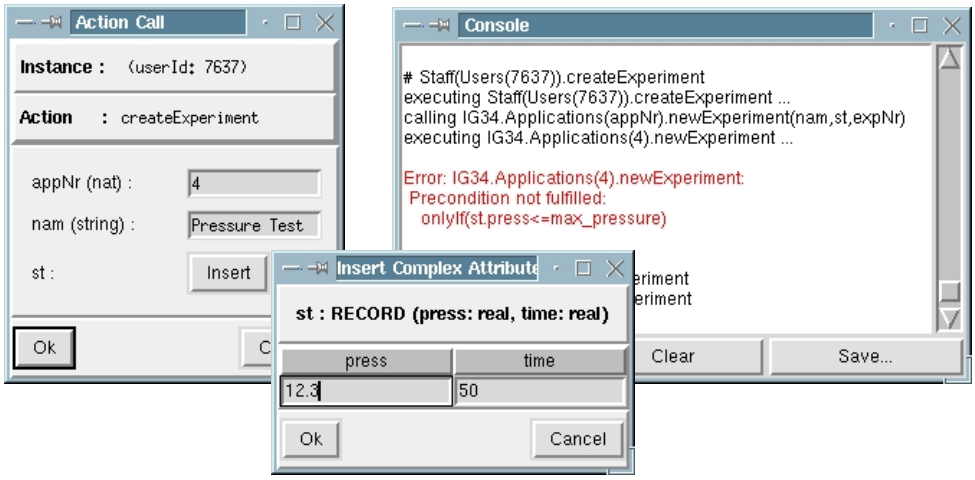


Figure 7.16: *trlanim* – Action Call Window of `createExperiment` I

requirement of the CATC system is that a staff user may not set up an experiment with a pressure greater than the maximum pressure determined in the application. So we are going to check if the specification meets this requirement. We give as value for the input parameter `applNr` the parameter value of the application we created before. Values with complex data types, i. e. records, lists, sets, bags and maps, are introduced in special windows. The experiment is set up with a pressure of `12.3`, that is greater than the maximum pressure established for the application (`11.5`), and a time of `50`. When executing the action, the console window reports that the action calls the action `newExperiment` in the application and that its precondition (`st.press <= max_pressure`) is not satisfied. So the state transition may not take place. Actually, like testing an implementation, by animating a specification we cannot assure its correctness with respect to the requirements. For this, we should perform an exhaustive testing, i. e. testing the specification in all possible circumstances. Since this is not practicable, we must select a set of *significant* test cases or scenarios. The successful execution

of a significant test case increases our confidence in the correctness of the specification. A testing strategy consists in grouping the values of the input domain into classes such that the values of a class are expected to behave in the same way. We can then choose a single test case as representative of each class. For instance, in the validation of the requirement above, the values of the setup pressure can be divided into two classes: the class of values which are greater than the allowed maximum pressure and the class of values which are less or equal to this pressure. Additionally, we should test the specification at the boundary between these classes, i. e. when both pressures are the same. Next, we call the action with a setup pressure of `11.2` that is less than the maximum pressure determined by the application. This is depicted in Fig. 7.17. As can be seen in the console window, this time

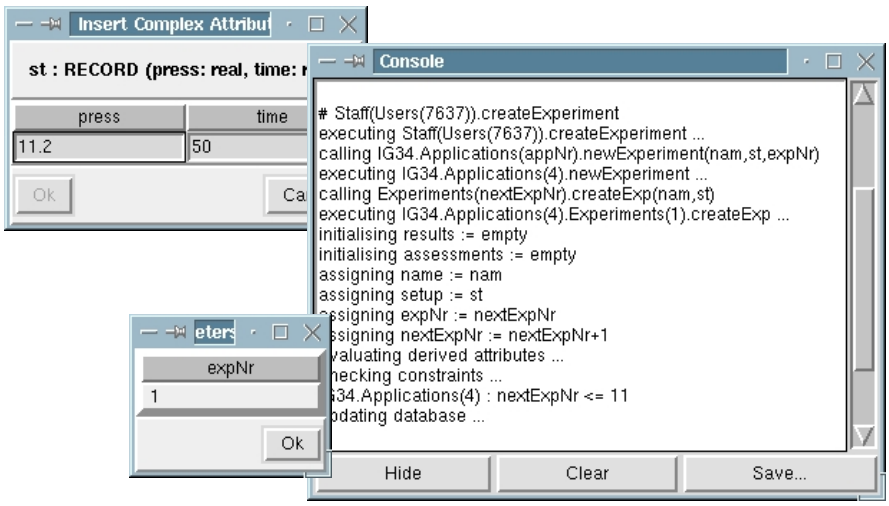


Figure 7.17: *trlanim* – Action Call Window of `createExperiment` II

the action is successfully executed. The action `createExperiment` of the staff user calls the action `newExperiment` of the application which in turn calls the birth action `createExp` of the new experiment. Attributes are then assigned. Once the behaviour of the actions has been executed and conforming with the execution model presented in the previous chapter, derived attributes are evaluated, integrity constraints are checked and the new attribute values are stored in the database. If the action called by the user of the animator has output parameters, their values are visualised in a window. For instance, the

output parameter `expNr` returns the identifier number of the new experiment (`1`). Now, we can see if the experiment has been successfully created and that its attributes have the expected values. From the instance view window of the object `IG34` (see Fig. 7.13), we select the component `Applications(4)`. We can then observe the interface of this object, as illustrated in Fig. 7.18. Object-valued attributes are visualised in separate windows. Clicking on the

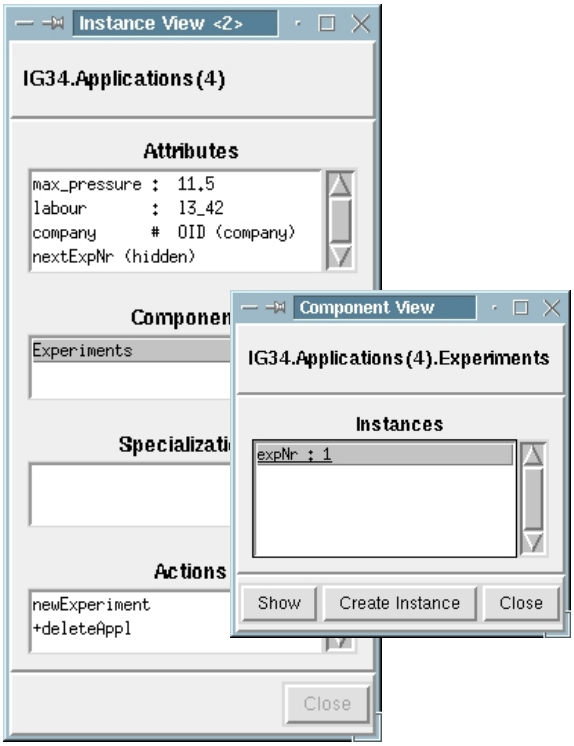


Figure 7.18: *trlanim* – Instance View Window of `Application`

attribute `company` pops up a window displaying the attributes of the company (see left window of Fig. 7.14). The attribute `nextExpNr` is not visible in the object's interface because it is declared as *hidden*. Clicking on the component `Experiments` shows the identifiers of the existing experiments. Here we can observe that the application has an experiment with the same identifier that the experiment created previously by the staff user. Double clicking on the experiment identifier and the object's interface of the experiment appears. This is depicted in Fig. 7.19. We can see that the values of the attributes

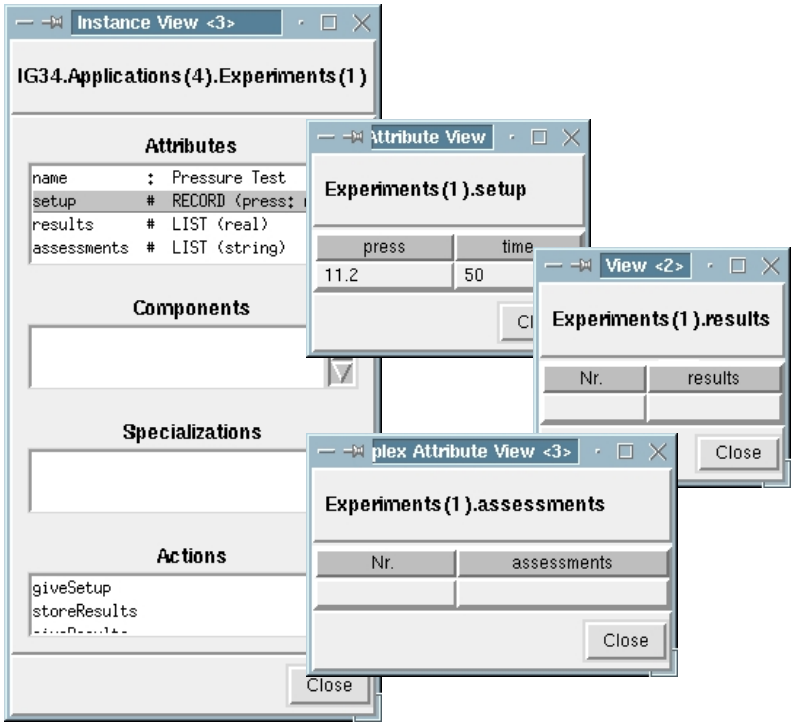


Figure 7.19: *trlanim* – Instance View Window of `Experiment`

`name` and `setup` correspond to those introduced in the action that created the experiment. As object-valued attributes, complex attributes are visualised in separate windows. In case of lists, an extra field (`Nr.`) indicates the position of each element in the list. As declared in the specification, the attributes `results` and `assessments` are initialised empty. In the same way we have validated the setup of an experiment by a staff user, we can validate the specification in other scenarios. For instance, a scenario would consists in an operator user reading the setup, starting the experiment and then storing the results. Another possible scenario would be then the reading of the results by a staff user and then the storing of his/her assessments.

The configuration window of *trlanim* is depicted in Fig. 7.20. If the animator is called from *trlbench*, the first check button sets if the animation should be automatically constructed, i. e. automatic calling, if necessary, to the checkers and the database and C++ code generators. The second check

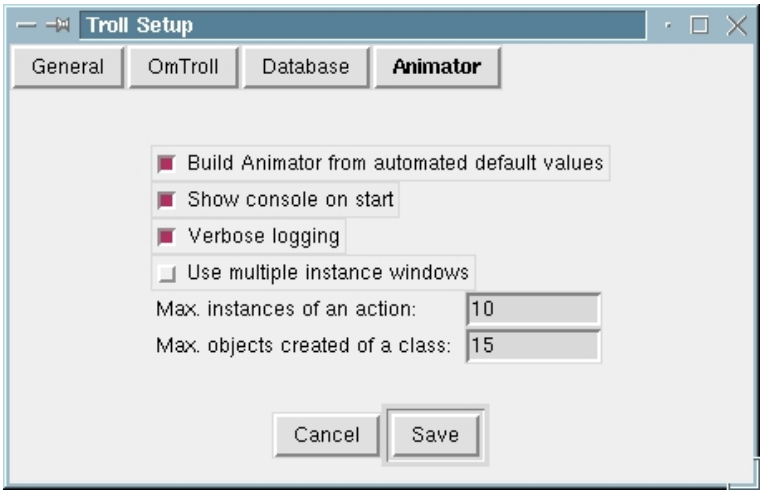


Figure 7.20: *trlanim* – Configuration Window

button shows/hides the console window on start. The third check button sets on/off verbose information. If the verbose mode is on, the execution trace is shown on the console. With the fourth check button, users can determine that at most one instance window is shown on the screen. Otherwise, a window is shown for each selected object. The former avoids displaying many windows on the screen. The latter is useful when comparing the interface of several objects. To assure the finiteness of state transitions, users may set the maximum number of instances of an action as well as the maximum number of objects created of a class that are allowed during a state transition.

trlanim has been implemented in C++ and Tcl/Tk. Information about the implementation of an initial version of the user interface can be found in [Sch99].

7.3 Summary

The TROLL workbench is a collection of software tools that support developers during the construction and validation of TROLL specifications. In this chapter, after a short description of the workbench architecture, we have presented the functionalities of each tool. The workbench may assist developers in the following way:

- A projects management tool helps developers to manage the creation of specification projects and the invocation of the TROLL tools.

- The specification can be modelled at a very abstract level using an OMTROLL graphical editor. OMTROLL diagrams are automatically translated into textual TROLL. The textual specification is then completed using a text editor. For this, a TROLL language mode has been implemented for the (X)Emacs editors. The language mode customises and extends these editors for supporting the edition of TROLL files. A reverse generation of OMTROLL diagrams from the textual specification is also possible.

- A checker assures that the syntax and static semantics of the specification are correct. The abstract syntax graph generated by the checker is used as the internal representation of the specification. A common graph interface allows the TROLL tools an easy and quick access to any information about the specification.

- An HTML code generator allows hypertext navigation through the specification components and the introduction of informal comments to document the model.

- The specification can be validated against user requirements through its animation. A database and a C++ code generator build the animation. The database generator creates a relational database schema which will contain the states of the objects created during the animation. The animator accesses the database through a persistence layer. The C++ code generator transforms the specification into a C++ library. Specifications are then animated in the TROLL animator. Here, users can observe, experiment and test the dynamic properties of the specification by its animation in different scenarios. The animator has an user-friendly interface that facilitates the participation of end users in the validation process.

As mentioned in Chapter 2, the modelling and validation of the specification are an iterative process. The specification can be animated in any modelling stage to see if it meets the expected behaviour. Errors or misunderstandings discovered during the animation entail changes in the specification which is modified in the editors accordingly.

The workbench has been designed to facilitate the implementation and introduction of new tools. Especially of interest are tools to support users in the design of animation scenarios as well as the automatic generation of scenarios from the specification.

Chapter 8

Conclusions

This thesis has presented an approach and toolset to the construction and animation of conceptual models specified with TROLL. In this chapter, we sum up the main contributions of the thesis and give some suggestions for further work.

8.1 Summary

The research in this work was motivated by the observation that although formal approaches to software specification provide more precise specifications and a basis for formal verification, there is a need for techniques to support the validation of these specifications against the informal user requirements. The main problem of requirements validation may be expressed by the aphorism, "One cannot go from the informal to the formal by formal means". Formal methods can prove that an implementation satisfies a specification, but they cannot prove that the specification captures a user's intuitive understanding of the system. It has been argued that the complexity of formal specifications makes them incomprehensible to users and, hence, it represents a severe obstacle to user validation. However, there are several ways of making a formal specification more accessible to users. One way consists in animating the specification. Through the execution of the specification, users can observe its dynamic properties in different scenarios to see if it adequately captures their real needs. In this context, our objective has been to give animation support for the formal object-oriented specification language TROLL. We have established the bases necessary for the construc-

tion of an animation environment for TROLL specifications and developed a prototype of the environment.

Before animating a specification, several analysis must assure that the syntax and static semantics of the specification are correct. To this end, we have analysed each part of a TROLL specification and established a set of static semantics rules to be checked during the semantic analysis. The syntax analyser creates an internal data structure containing the syntax tree of the specification. We have designed this structure as a directed labelled attributed graph. Since the graph only stores the abstract syntax of the specification, it is smaller and easier to manage in the subsequent phases. Furthermore, unnecessary vertexes' are eliminated from the graph. The graph is extended to include context-sensitive information by the semantic analyser. Context-sensitive information is not stored, as usual, in form of attributes in the vertexes, but by extending the graph with new edges and vertexes. In this way, vertexes have a uniform structure without the need of containing unnecessary attributes. Furthermore, if new information is required to be stored in the graph, it is not necessary to change the vertexes structure but just extend the graph with new edges and vertexes. The graph is used as the internal representation of the specification in the animation environment. Tools access the information contained in the graph through a common interface which hides details from the graph structure.

Objects created during the animation are persistent. This allows users to interrupt anytime the animation of the specification in a scenario and to continue it in the same scenario later. Furthermore, objects created in an animation session can be reused in other sessions. This is especially helpful if the scenarios in which the specification is animated require the creation of a large number of objects. In the TROLL animation environment, objects are stored in an RDBMS. Besides data independence and integrity, an advantage of using a DBMS is that it allows the concurrent access to the data. So users can animate the same object society simultaneously. In order to generate the database schemas that hold the state of the objects created during the animation, we have analysed how the static structure of TROLL objects can be mapped into relational tables. We have presented rules for mapping object identifiers, objects, components, specialisations and attributes into database schemas. The mapping of TROLL concepts into the relational model is not straightforward. Nevertheless, the access to the database is encapsulated by a persistence layer. So the animator does not know about how objects are stored in the database. The animator just gives the global identifier of the

object and the required operation, e. g. creation, deletion, read or update of an attribute, to the persistence layer which then performs the operation.

In order to execute the specifications, we have first analysed the execution of state transitions in TROLL. A state transition is triggered by the calling to an initial action. This action may in turn call other actions in the same or in other objects establishing an action chain to be executed synchronously. We have addressed the issues of parallel execution, conflicts in attribute and variable assignments, consistency, termination and atomicity. Since all actions involved in a calling relation are conceptually atomic in duration, a correct execution order, which considers data flow dependences, must be determined. An execution model of state transitions in a sequential environment has been given. During the execution of a state transition, run-time checks assure that the number of actions in the calling relation is finite and that all actions may take place, i. e. all action preconditions are fulfilled, no attribute or variable is assigned with different values and no integrity constraint would be violated in the new state.

Next, we have presented the implementation of the execution model in the animator and the structure of the C++ code to be generated from the specifications. The class structure has been maintained in the generated code. So changes in the specification classes only require the re-compilation of the corresponding C++ classes. Furthermore, if TROLL classes are reused in other systems, the corresponding C++ classes can be reused as well.

Finally, we have presented the TROLL workbench, a collection of software tools to support the modelling and validation of TROLL specifications. A graphical front-end allows developers to manage the creation of specification projects and the simultaneous use of the tools. Specifications are first modelled using a graphical OMTROLL editor. OMTROLL diagrams are automatically translated into a textual TROLL specification which can be refined using a text editor. A reverse generation of OMTROLL diagrams from the textual specifications is also possible. A checker assures that the syntax and static semantics of the specification are correct and creates an abstract syntax graph from the specification which is used as internal data structure by the remaining tools. A documentation tool generates HTML code from the specification which can be browsed in an HTML-browser allowing hypertext navigation through the specification components. A database generator and a C++ code generator build the animation. Specifications can then be executed in a TROLL animator. In the animator, users can observe the state of the objects, navigate through their relationships, simulate the occurrence

of events and observe the execution traces. In this way, they can investigate the adequacy of the specification in a variety of scenarios and increase their understanding of the intended behaviour of the system. The facts that the workbench is a collection of stand-alone applications which are integrated through a common graphical front-end, and the data structures are accessed through abstract interfaces hiding implementation details facilitate the modification of the tools and the incorporation of new ones.

In summary, the contribution of this thesis is a systematic approach and workbench environment to support the incremental construction and validation through animation of formal specifications. Our approach maximises the strengths of formal specifications (e. g. conciseness and unambiguity) and rapid system prototyping (e. g. risk management and early user involvement).

Some of the results prior to this thesis has been published in [Gra97b, Gra97a, GKE97, GK97, Gra98, KG98, GKK+98] and presented in several talks, especially in workshops of the European project ASPIRE. The TROLL workbench has been shown in several demos. The workbench has been started being used by students doing their diploma theses in cooperation projects with PTB. Some positive experiences have already been reported in [KG98, Kan99, Win00, Sch00]. Currently, we intend to port the workbench to Windows, so it can be used directly at the computer labs from PTB. The workbench is also being used by the author himself in the specification of a railway traffic control application in the context of a DFG project. The animator has significantly helped us to detect errors and misunderstandings in the specification.

8.2 Further Work

Possible directions for future work include the following issues:

- *Graphical visualisation of execution traces:* Currently, the animator shows execution traces in a textual form. A way of making execution traces more intuitive and easier to understand consists in their representation using graphical notations, for instance, in the form of sequence diagrams. This is especially helpful for end users who may have problems in understanding the textual representation. Furthermore, if requirements have been elicited using graphical scenario-based techniques, the visualisation of execution traces in a similar notation

would facilitate the validation of the models by allowing the comparison of execution traces and scenarios.

- *Processing of batch files:* The animator can easily be extended to support batch execution. We can prepare the relevant sequence of events in a batch file and ask the animator to execute steps iteratively, reading in the events from the file. In this way, we can re-run pre-established scenarios. Moreover, this is a prerequisite for the animation of scenarios constructed with other tools such as scenario editors and scenario generation tools.

- *Support for scenario construction:* Methodological and tool support for the construction of scenarios has to be considered. Requirements could be captured using scenario techniques. These scenarios would be used as a starting point for the elaboration of the specification. Once the specification has been built, the same scenarios would also be used for validating the specification in the animator. Since the scenarios and the specification are directly related, we could add a scenario diagram editor in the OmTroll editor, so the construction of both scenarios and specifications would be integrated in a tool.

- *Automated scenario generation:* A future research direction is the automated generation of animation scenarios from the specification. Automated specification-based software testing is an emerging discipline which aims at obtaining test cases from the specification in order to test the implementation [Pos96]. A test case generator analyses the specification and uses test design strategies such as functional testing and boundary value analysis to create test cases. In our context, the generated test cases would be used already in the requirements phase for testing the specification in the animator. Of course, the same tests can be reused later when testing the final implementation to ensure that the implementation corresponds to the specification.

- *Multiuser support:* The workbench could be extended to support the development of specification projects by a group of people working together. At the modelling level, an interface to a version control system tool such as CVS[1] could be constructed to allow the versioning and

[1]CVS (Concurrent Versions System) is a free software tool supporting version

shared access to the specification files. At the validation level, the animator could be extended to support cooperative animation sessions in which every participant plays the role of one or several objects. In this case, mechanisms for allowing a distributed execution of the specification in several machines must be studied.

- *Evolutionary prototyping:* In the workbench, code is generated for animation purposes, i. e. it is not as efficient as the required final code and includes control code specific for the animation such as code for checking termination conditions and showing execution traces. Nevertheless, transformation rules used in the code generation could be adopted in the design and implementation phases, and some parts of the prototype code could be evolved into the final application. These issues need to be addressed in the future.

- *Support for* TROLL *module concepts:* Currently, TROLL is being extended by module constructs with the purpose to support in-the-large specifications and reuse of specification parts [Eck98]. The workbench should be adapted accordingly. Furthermore, the module constructs would be helpful for structuring and reusing parts of the data managed in the workbench such as the specification files created in the editors, the graphs generated by the syntax and static semantics checker and the code generated for the animation.

- *Integration of a model-checking tool:* A model-checking approach to the verification of system properties in TROLL specifications is currently under investigation [EP00]. A model-checker for TROLL could easily be integrated into the workbench. The model-checker would make use of the existing tools, such as editors, syntax and static semantics checkers and the syntax graph interface. The integration of a model-checker into the workbench would make the workbench a very complete tool addressing two of the most important aspects in the construction of correct software specifications: validation of informal requirements and verification of system properties.

management and concurrent editing of source files. Its web page is located at http://www.gnu.org/software/cvs/cvs.html

Bibliography

[AB91] M. C. Atkins and A. W. Brown. Principles of Object-Oriented Systems. In J. A. McDermid, editor, *Software Engineer's Reference Book*, chapter 39. Butterworth Heinemann, 1991.

[ACPT99] P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Database Systems: Concepts, Languages and Architectures*. McGraw-Hill, 1999.

[Agr86] W. W. Agresti, editor. *New Paradigms in Software Development*. IEEE Computer Society Press, 1986.

[AJ96] K. Arnold and Gosling J. *The Java Programming Language*. Sun Microsystems. Addison-Wesley, 1996.

[Ale93] M. Alexander. *Implementierung eines Term- und Formelauswertes für TROLL-Light Spezifikationen*. Diploma Thesis, Technische Universität Braunschweig, September 1993.

[Alm94] G. Almgren. *TBENCH Graphical Specification Manager*. Diploma Thesis, Technische Universität Braunschweig, 1994.

[Amb99] S. W. Ambler. Mapping Objects to Relational Databases. White Paper, 1999. Available at htttp://www.ambysoft.com/mappingObjects.html.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Bal85] R. Balzer. A 15-Year Perspective on Automatic Programming. *IEEE Transactions on Software Engineering*, 11(11):1257–1268, November 1985.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System - The Story of O_2*. Morgan Kaufmann, San Francisco, CA, 1992.

[BEM92] A. Brown, A. Earl, and J. McDermid. *Software Engineering Environments - Automated support for Software Engineering*. Series in Software Engineering. McGraw-Hill International, 1992.

[BH95] J. P. Bowen and M. G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(4):34–41, July 1995.

[BK96] A. Brenecke and R. Keil-Slawik, editors. *History of Software Engineering*. Position Papers for Dagstuhl Seminar 9635, August 1996. Available at http://www.dagstuhl.de/DATA/Reports/9635.

[BLR99] R. Berghammer, Y. Lakhnech, and W. Reif. Formal Methods and Tools: Introduction and Overview. In *Tool Support for System Specification, Development and Specification*, pages 1–17. Springer-Verlag/Wien, 1999.

[BM93] E. Bertino and L. Martino. *Object-Oriented Databases Systems: Concepts and Architectures*. Addison-Wesley, 1993.

[BMP92] D. Bell, I. Morrey, and J. Pugh. *Software Engineering: A Programming Approach*. Prentice-Hall International (UK), 2nd edition, 1992.

[Boe81] B. W. Boehm. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981.

[Boe88] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *Computer*, pages 61–72, 1988.

[Boo93] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjaming/Cummings, Redwood City, CA, 1993.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The GemStone Object Database Management System. *Communications of the ACM*, 34(10):64–77, 1991.

[BP94] L. Bunge and M. Probst. *Entwurf und Implementierung des TBench Repositories (TREP).* Pre-diploma Thesis, Technische Universität Braunschweig, October 1994.

[Bri93] J. Brinckmann. *Implementierung des TROLL-Light Ausführungsmodells.* Diploma Thesis, Technische Universität Braunschweig, September 1993.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Object Technology Series. Addison-Wesley, 1998. The UML home site is located at: http://www.rational.com/uml.

[BW95] K. Brown and G. Whitenack. Crossing Chasms, A Pattern Language for Object-RDBMS Integration. White Paper, Knowledge Systems Corp., 1995. Available at http://www.ksccary.com/articles.htm.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, S. Dollin, Hayes Gilchrist, F. Hayes, and P. Jeremes. *Object-Oriented Development - The Fusion Method.* The Object-Oriented Series. Prentice Hall, Englewood Cliffs, NJ, 1994.

[CD94] S. Cook and J. Daniels. *Designing Object Systems - Object-Oriented Modeling with Syntropy.* The Object-Oriented Series. Prentice Hall, Englewood Cliffs, NJ, 1994.

[Cer98] P. E. Cerucci. *A History of Modern Computing.* Massachusetts Institute of Technology, 1998.

[CGH92] S. Conrad, M. Gogolla, and R. Herzig. TROLL *light*: A Core Language for Specifying Objects. Informatik-Bericht 92–02, Technische Universität Braunschweig, 1992.

[Con94] S. Conrad. *Ein Basiskalkül für die Verifikation von Eigenschaften synchron interagierender Objekte.* Fortschritt-Berichte Reihe 10, Nr. 295. VDI-Verlag, Düsseldorf, 1994.

[CRS90] E. Cusack, S. Rudkin, and C. Smith. An Object-Oriented Interpretation of LOTOS. In S. Vuong, editor, *Formal Description*

Techniques II, FORTE'89, pages 211–226. North–Holland, Amsterdam, 1990.

[CSS89] J.-F. Costa, A. Sernadas, and C. Sernadas. *OBL-89 Users Manual. Internal Report.* INESC, Lisbon, 1989.

[CW96] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. In *ACM Workshop on Strategic Directions in Computing, MIT*, June 1996.

[Dal92] H. Dalianis. A Method for Validating a Conceptual Model by Natural Language Discourse Generation. In P. Loucopoulos, editor, *Proc. 4th Int. Conf. on Advanced Information Systems Engineering (CAiSE'92)*, pages 425–444. Springer, Berlin, LNCS 593, 1992.

[Dan95] C. Danker. *Transformation von TROLL-Objektbeschreibungen in Schemata relationaler Datenbanken.* Diploma Thesis, Technische Universität Braunschweig, 1995.

[Dat95] C. Date. *An Introduction to Database Systems.* Addison-Wesley, 1995.

[DB95] P. Du Bois. *The Albert II Language. On the Design and the Use of a Formal Specification Language for Requirements Analysis.* PhD thesis, University of Namur (Belgium), 1995.

[DB97] P. Du Bois. The Albert II Reference Manual. Technical report, University of Namur (Belgium), 1997. Available at http://www.info.fundp.ac.be/~phe/albert.html.

[DDD94] E. DuBois, P. Du Bois, and F. Dubru. Animating Formal Requirements Specifications of Cooperative Information Systems. In M.L. Brodie, M. Jarke, and M.P. Papazoglou, editors, *Proc. 2nd Int. Conf. Cooperative Information Systems (CoopIS'94)*, pages 101–112, 1994.

[DDH72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, New York, 1972.

[DH97] G. Denker and P. Hartel. TROLL – An Object Oriented Formal Method for Distributed Information System Design: Syntax and Pragmatics. Informatik-Bericht 97–03, Technische Universität Braunschweig, 1997.

[DK92] E.H. Dürr and J.v. Katwijk. VDM++, A Formal Specification Language for Object–Oriented Design. In *Proceedings of TOOLS7 (Technology of Object-Oriented Languages and Systems)*. Prentice Hall, 1992.

[DMN67] O.-J. Dahl, B. Myrhaug, and K. Nygaard. SIMULA 67, Common Base Language. Technical report, Norwegian Computer Center, Oslo, 1967.

[EC00] H.-D. Ehrich and C. Caleiro. Specifying communication in distributed information systems. *Acta Informatica*, 36(Fasc. 8):591–616, 2000.

[Eck98] S. Eckstein. Towards a Module Concept for Object Oriented Specification Languages. In J. Barzdins, editor, *Proc. 3rd Int. Baltic Workshop on Data Bases and Information Systems, Riga, Latvia, April 15-17*, volume 2, pages 180–188. Institute of Mathematics and Informatics, University of Latvia, Latvian Academic Library, 1998.

[ECSD98] H.-D. Ehrich, C. Caleiro, A Sernadas, and G. Denker. Logics for Specifying Concurrent Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 167–198. Kluwer Academic Publishers, 1998.

[EDS93] H.-D. Ehrich, G. Denker, and A. Sernadas. Constructing Systems as Object Communities. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. Theory and Practice of Software Development (TAPSOFT'93)*, pages 453–467. Springer, Berlin, LNCS 668, 1993.

[EG01] S. Einer and A. Grau. Integrating Petri Nets and TROLL in the Modeling of Engineering Systems. In *Proc. of the 2nd Workshop on Formal Specifications of Computer Based Systems (FSCBS'01) Washington DC*, April 2001. *To appear.*

[EGS90] H.-D. Ehrich, J. A. Goguen, and A. Sernadas. A Categorial Theory of Objects as Observed Processes. In J.W. deBakker, W.P. deRoever, and G. Rozenberg, editors, *Proc. REX/FOOL Workshop*, pages 203–228, Noordwijkerhood (NL), 1990. LNCS 489, Springer, Berlin.

[EH96] H.-D. Ehrich and P. Hartel. Temporal Specification of Information Systems. In A. Pnueli and H. Lin, editors, *Logic and Software Engineering, Proc. Int. Workshop in Honor of C.S. Tang,Beijing, 14-15 August 1995*, pages 43–71. World Scientific, 1996.

[Ehr99] H.-D. Ehrich. Object specification. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, chapter 12, pages 435–465. Springer, 1999.

[EJDS94] H.-D. Ehrich, R. Jungclaus, G. Denker, and A. Sernadas. Object-Oriented Design of Information Systems: Theoretical Foundations. In J. Paredaens and L. Tenenbaum, editors, *Advances in Database Systems, Implementations and Applications*, pages 201–218. Springer Verlag, Wien, CISM Courses and Lectures no. 347, 1994.

[ELL94] R. Elmstrøm, P. G. Larsen, and P. B. Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM SIGPLAN*, 29(9):77–81, 1994. The IFAD web site is located at http://www.ifad.dk.

[Emb94] U. Embshoff. *Ein sprachsensitiver Editor für objektorientierten Spezifikationen.* Diploma Thesis, Technische Universität Braunschweig, 1994.

[Eng86] G. Engels. *Graphen als Zentrale Datensstrukturen in einer Software -Entwicklungsumgebung.* PhD thesis, Mainz, VDI - Verlag, Reihe 10: informatik Kommunikationstechnik, Nr. 62, 1986.

[EP00] H.-D. Ehrich and Ralf Pinger. Checking Object Systems via Multiple Observers. In *International ICSC Congress on Intelligent Systems & Applications (ISA'2000)*, volume 1, pages 242–

248. University of Wollongong, Australia, International Computer Science Convetions (ICSC), Canada, 2000.

[ES90] H.-D. Ehrich and A. Sernadas. Algebraic Implementation of Objects over Objects. In J. W. deBakker, W.-P. deRoever, and G. Rozenberg, editors, *Proc. REX Workshop "Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness"*, pages 239–266. LNCS 430, Springer, Berlin, 1990.

[Esp93] Espírito Santo Data Infomática, Lisbon. *OBLOG CASE V. 1.0 - The User's Guide*, 1993.

[Fuc92] N.E. Fuchs. Specifications Are (Preferably) Executable. *Software Engineering Journal*, 7(5):323–334, 1992.

[Fus97] M. L. Fussell. Foundations of Object Relational Mapping. White Paper; ChiMu Corp., 1997. Available at http://www.chimu.com/publications/objectRelational/.

[GCD+95a] M. Gogolla, S. Conrad, G. Denker, R. Herzig, and N. Vlachantonis. A Development Environment for an Object Specification Language. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):505–508, June 1995.

[GCD+95b] M. Gogolla, S. Conrad, G. Denker, R. Herzig, N. Vlachantonis, and Ehrich H.-D. TROLL *light* — The Language and its Development Environment. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software, Final Report*, pages 205–220. Springer, 1995. LNCS 1009.

[Ger00] A. Gerstberger. *Entwurf und Implementierung eines HTML-Generators für die Dokumentation von* TROLL *Spezifikationen*. Pre-diploma Thesis, Technische Universität Braunschweig, March 2000.

[GJM91] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[GK97] A. Grau and M. Kowsari. A Validation System for Object-Oriented Specifications of Information Systems. In R. Manthey

and V. Wolfengangen, editors, *Proc. of the First East-European Symposium on Advances in Databases and Information Systems (ADBIS'97) St. Petersburg, September 1997*. eWiC, Springer, 1997.

[GKE97] A. Grau, M. Kowsari, and H.-D. Ehrich. A CASE-Tool Environment for Developing and Validating Conceptual Models. *9th Conference on Advanced Information Systems Engineering (CAISE'97), Barcelona, June 18-20 1997*, poster session, 1997.

[GKK+98] A. Grau, J. Küster Filipe, M. Kowsari, S. Eckstein, R. Pinger, and H.-D. Ehrich. The TROLL Approach to Conceptual Modelling: Syntax, Semantics and Tools. In T.W. Ling, S. Ram, and M.L. Lee, editors, *Proc. of the 17th Int. Conference on Conceptual Modeling (ER'98), Singapore* , pages 277–290. Springer, LNCS 1507, November 1998.

[Gla98] R. L. Glass. *Software Runaways*. Prentice Hall, 1998.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[Gra97a] A. Grau. An Animation System for Validating Object-Oriented Conceptual Models. In J.P. Tolvanen and A. Winter, editors, *4th Doctoral Consortium on Advanced Information Systems Engineering (CAISE'97), Barcelona, June 16-17 1997*. Fachberichte Informatik 14/97, Univ. Koblenz-Landau, 1997.

[Gra97b] A. Grau. Validating Object-Oriented Specifications through Animation. In C. Eckert, T. Polle, and T. Stülten, editors, *9. Workshop Grundlagen von Datenbanken , Königslutter, Mai 20-23 1997*, pages 26–30. Forschungsbericht Nr. 643, Fachbereich Informatik, Univ. Dortmund, 1997.

[Gra98] A. Grau. Analysing Object Specifications for Execution. In M-H. Scholl, H. Riedel, T. Grust, and D. Gluche, editors, *10. Workshop Grundlagen von Datenbanken , Konstanz, June 2-5 1998*, pages 32–36. Konstanzer Schriften in Mathematik und Informatik Nr. 63, Univ. Konstanz, 1998.

[Gul96] J.A. Gulla. A General Explanation Component for Conceptual Modeling in Case Environments. *ACM Transactions on Information Systems*, 14(3):297–329, 1996.

[Hal90] A. Hall. Seven Myths of Formal Methods. *IEEE Software*, pages 11–19, September 1990.

[Har92] D. Harel. Biting the Silver Bullet - Towards a Brighter Future of Systems Development. *IEEE Computer*, 25(1):8–20, January 1992.

[Har95] T. Hartmann. *Entwurf einer Sprache für die verhaltensorientierte konzeptionelle Modellierung von Informationssystemen*, volume 1 of *Reihe DISDBIS*. infix-Verlag, Sankt Augustin, 1995.

[Har97a] P. Hartel. *Konzeptionelle Modellierung von Informationssystemen als verteilte Objektsysteme*. Reihe DISDBIS. infix-Verlag, Sankt Augustin, 1997.

[Har97b] J. Hartmann. *Entwurf und Implementierung eines grafischen Editors für objektorientierte Spezifikationen*. Diploma Thesis, Technische Universität Braunschweig, December 1997.

[HD98] P. Heymans and E. Dubois. Scenario-Based Techniques for Supporting the Elaboration and the Validation of Formal Specifications. *Requirements Engineering Journal, Springer-Verlag*, 3:202–218, 1998.

[HDK+97] P. Hartel, G. Denker, M. Kowsari, M. Krone, and H.-D. Ehrich. Information Systems Modelling with TROLL: Formal Methods at Work. *Information Systems*, 22(2-3):79–99, 1997.

[Her95] R. Herzig. *Zur Spezifikation von Objektgesellschaften mit TROLL light*. Fortschritt-Berichte Reihe 10, Nr. 336. VDI-Verlag, Düsseldorf, 1995.

[Hey97] P. Heymans. The Albert II Specification Animator. Technical report, CREWS Report Series 97-13, August 1997. Available at http://sunsite.informatik.rwth-aachen.de/CREWS.

[HG94] R. Herzig and M. Gogolla. An Animator for the Object Specification Language TROLL *light*. In V.S. Alagar and R. Missaoui, editors, *Proc. Colloquium on Object Orientation in Databases and Software Engineering (COODBSE'94)*, pages 4–17. Université du Quebéc à Montréal, 1994.

[HJ89] I.J. Hayes and C.B. Jones. Specifications Are not (Necessarily) Executable. *Software Engineering Journal*, 4(6):330–338, 1989.

[HJS93] T. Hartmann, R. Jungclaus, and G. Saake. Animation Support for a Conceptual Modelling Language. In V. Mařík, J. Lažanský, and R.R. Wagner, editors, *Proc. 4th Int. Conf. on Database and Expert Systems Applications (DEXA), Prague*, pages 56–67. LNCS 720, Springer, Berlin, 1993.

[HJSE92] T. Hartmann, R. Jungclaus, G. Saake, and H.-D. Ehrich. Spezifikation von Objektsystemen. In R. Bayer, T. Härder, and P.C. Lockemann, editors, *Objektbanken für Experten*, pages 220–242. Springer, Berlin, Reihe Informatik aktuell, 1992.

[HKSH94] T. Hartmann, J. Kusch, G. Saake, and P. Hartel. Revised Version of the Conceptual Modeling and Design Language TROLL. In R. Wieringa and R. Feenstra, editors, *Working papers of the International Workshop on Information Systems - Correctness and Reusability*, pages 89–103. Vrije Universiteit Amsterdam, 1994.

[HS93] T. Hartmann and G. Saake. Abstract Specification of Object Interaction. Informatik-Bericht 93–08, Technische Universität Braunschweig, 1993.

[HSJ+94] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kusch. Revised Version of the Modelling Language TROLL (Version 2.0). Informatik-Bericht 94–03, Technische Universität Braunschweig, 1994.

[IEE84] IEEE Guide to Software Requirements Specifications. ANSI/IEEE Std 830-1984. The Institute of Electrical and Electronics Engineers, Inc., 1984.

[ISO82] Concepts and Terminology for the Conceptual Schema and the Information Base. Report No 695, ISO TC97/SC5/WG3, 1982.

[IYIK96] H. Ishikawa, Y. Yamane, Y. Izumida, and N. Kawato. An Object-Oriented Database System Jasmine: Implementation, Application, and Extension. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):285–303, April 1996.

[JC92] L. Jesus and R. Carapuça. Automatic Generation of Documentation for Information Systems. In *CAISE'92*, pages 48–64, 1992.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. Foundations of Specification Animations. In *mural: A Formal Developmen Support System*, chapter 9. Springer-Verlag London, 1991.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, London, 2nd edition, 1990.

[JSHS91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91-04, Technische Universität Braunschweig, 1991.

[JSHS96] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL – A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems*, 14(2):175–211, April 1996.

[JWH+94] R. Jungclaus, R.J. Wieringa, P. Hartel, G. Saake, and T. Hartmann. Combining TROLL with the Object Modeling Technique. In B. Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen. GI-Fachgespräch FG 1: Integration von semi-formalen und formalen Methoden für die Spezifikation von Software*, pages 35–42. Springer, Informatik aktuell, 1994.

[Kan99] M. Kananian. *Objektorientierte Erweiterung eines Informationssystemes zur Konstruktionsprüfung explosionsgeschützer elektrische Betriebsmittel*. Diploma Thesis, Technische Universität Braunschweig, May 1999.

[Kel97] W. Keller. Mapping Objects to Tables: A Pattern Language. In *Proc. of the 1997 European Pattern Languages of Programming Conference, Irrsee, Germany*. Siemens Technical Report 120/SW1/FB, 1997.

[KG98] M. Kowsari and A. Grau. An Evaluation of an Object Oriented Formal Method for Specifying Information Systems. In K. Siau, editor, *Proceedings of the Third CAiSE/IFIP 8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD'98). Pisa, Italy. June 8-9, 1998*, pages M1–M12. University of Nebraska-Lincoln, USA, 1998.

[KHHS95a] J. Kusch, P. Hartel, T Hartmann, and G. Saake. Gaining a Uniform View of Different Integration Aspects in a Prototyping Environment. In *Proc. 6th Int. Conf. on Database and Expert Systems Applications (DEXA'95)*, pages 38–47. Springer Verlag, Berlin, LNCS 978, 1995.

[KHHS95b] J. Kusch, P. Hartel, T. Hartmann, and G. Saake. Integration einer Prototyping-Umgebung durch Objektorientierte Spezifikation. Preprint 5, Univ. Magdeburg, Fakultät für Informatik, 1995.

[KK88] J. Kramer and N. Keng. Animation of Requirements Specifications. *Software – Practice and Experience*, 18(8):749–774, August 1988.

[Kle94] S. Klein. *TBench Integration Manager - ein Integrationsrahmen für eine Prototyping-Umgebung*. Diploma Thesis, Technische Universität Braunschweig, November 1994.

[Kne97] R. Kneuper. Limits of Formal Methods. *Formal Aspects of Computing*, 9:379–394, 1997.

[Kow96] M. Kowsari. Formal Object Oriented Specification Language TROLL in Information System Design. In H.-M. Haav and B. Thalheim, editors, *Doctoral Consortium of 2nd Int. Baltic Workshop on Databases and Information Systems, Tallinn*, 1996.

[KSTM98] K. Koskimies, T. Systä, J. Tuomi, and T. Mänistö. Automated Support for Modeling OO Software. *IEEE Software*, 1998.

[Kus93] Jan Kusch. *Prototypimplementierung eines Animationssystems für objektorientierte Spezifikationen.* Diploma Thesis, Technische Universität Braunschweig, 1993.

[Küs00a] J. Küster Filipe. *Foundations of a Module Concept for Distributed Object Systems.* PhD Thesis, Technical University Braunschweig, September 2000.

[Küs00b] J. Küster Filipe. Fundamentals of a Module Logic for Distributed Object Systems. *Journal of Functional and Logic Programming*, 2000(3), March 2000.

[Lan91] K. Lano. Z++: an Object-Oriented Extension to Z. In J. Nicholls, editor, *Z Users Workshop: Proc. 4th Annu. Z User Meeting*, pages 151–172. Workshops in Computing. Springer-Verlag, Berlin, 1991.

[Lan95] K. Lano. From Analysis to Formal Specification. In *Formal Object-Oriented Development*, chapter 3. Springer Verlag, 1995.

[Let99] P. O. Letelier Torres. *Animación Automática de Especificaciones OASIS utilizando Programación Lógica Concurrente.* PhD thesis, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 1999.

[LL93] V. Lalioti and P. Loucopoulos. Visualisation for Validation. In C. Rolland, F. Bodart, and C. Cauvet, editors, *Proc. 5th Int. Conf. on Advanced Information Systems Engineering (CAiSE'93)*, pages 143–163. Springer, Berlin, LNCS 685, 1993.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreib. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, 1991.

[Loo95] M. E. S. Loomis. *Object Databases: the Essentials.* Addison-Wesley, 1995.

[Lou92] P. Loucopoulos. Conceptual Modeling. In P. Loucopoulos and R. Zicari, editors, *Conceptual Modeling, Databases and CASE, Sect. 1*, pages 1–26. John Wiley and Sons, Inc., 1992.

[LP92] P. G. Larsen and N. Plat. Standards for Non-Executable Specification Languages. *The Computer Journal*, 35(6):567–573, 1992.

[LRSP98] P. Letelier, I. Ramos, P. Sánchez, and O. Pastor. *OASIS versión 3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto*. Servicio de Publicaciones de la Universidad Politécnica de Valencia, SPUPV-98.4011, 1998.

[LRT92] K. Lodaya, R. Ramanujam, and P.S. Thiagarajan. Temporal Logics for Communicating Sequential Agents. *Int. Journal of Foundations of Computer Science*, 3(2):117–159, 1992.

[LSR97] P. Letelier, P. Sánchez, and I. Ramos. Animation of System Specifications using Concurrent Logic Programming. In *Symposium of Logical Approaches to Agent Modeling and Design (ESSLLI'97), Aix-in-Provence (France)*, 1997.

[LSR99] P. Letelier, P. Sánchez, and I. Ramos. Animation of Conceptual Models using Two Concurrent Environments: An Overview. In *Proc. of the 3rd IMACS/IEEE International Multiconference on Circuits, Systems, Communication and Computers, Greece*, 1999.

[Mel96] J. Melton. SQL3 Update. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 566–672, 1996.

[Mer94] M. Mertens. *TEA - Ein Werkzeug zur Analyse von TROLL Ausdrücken*. Pre-diploma Thesis, Technische Universität Braunschweig, October 1994.

[Mey92] B. Meyer. *Eiffel - The Language*. Prentice Hall, 1992.

[MJHB98] I. Morrey, Siddiqi J., R. Hibberd, and G. Buckberry. A Toolset to Support the Construction and Animation of Formal Specifications. *Systems and Software*, 41:147–160, 1998.

[MR91] J. McDermid and P. Rook. Software Development Process Models. In J. A. McDermid, editor, *Software Engineer's Reference Book*, chapter 16. Butterworth Heinemann, 1991.

[Muk95] P. Mukherjee. Computer-Aided Validation of Formal Specifications. *Software Engineering Journal*, pages 133–140, July 1995.

[Myl98] J. Mylopoulos. Information Modeling in the Time of the Revolution. *Information Systems*, 23(3/4):127–155, 1998.

[Ner95] G. Nerjes. *Ansätze zur Implemetierung der dynamischen Aspecte von TROLL-Spezifikationen.* Diploma Thesis, Technische Universität Braunschweig, November 1995.

[Neu96] K. Neumann. *Datenbank-Technik für Anwender.* Carl Hanser Verlag München Wien, 1996.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, part 1. *Theoretical Computer Science*, 13:85–108, 1981.

[Omo91] S. M. Omohundro. The Sather Language. Technical report, International Computer Science Institut (ISCI), Berkeley, California, 1991.

[O'N92] G. O'Neill. Automatic Transformation of VDM Specifications into Standards ML Programs. *The Computer Journal*, 35(6):623–624, 1992.

[OS96] A. Olivé and M.R. Sancho. Validating Conceptual Specifications through Model Execution. *Information Systems*, 21(2):167–186, 1996.

[Özc98] M. B. Özcan. Use of Executable Formal Specifications in User Validation. *Software-Practice and Experience*, 28(13):1359–1385, November 1998.

[PCR99] O. Pastor, J. H. Canós, and I. Ramos. From CASE to CARE (Computer-Aided Requirements Engineering). In J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, and E. Métais, editors, *Proc. of the 18th Int. Conference on Conceptual Modeling (ER'99), Paris (France)* , pages 278–292, November 1999.

[PIP+97] O. Pastor, E. Insfran, V. Pelechano, J. Romero, and J. Merseguer. OO-Method: An OO Software Production Environment Combining Conventional and Formal Methods. In A. Olive and J. A. Pastor, editors, *Proc. of the 9th International Conference on Advanced Information Systems Engineering (CAiSE'97), Barcelona*, pages 145–158, 1997.

[Pos96] R.M. Poston. *Automating Specification-Based Software Testing*. IEEE Computer Society, Los Alamitos, 1st edition, 1996.

[PPIG98] O. Pastor, V. Pelechano, E. Insfrán, and J. Gómez. From Object-Oriented Conceptual Modeling to Automated Programming in Java. In T.W. Ling, S. Ram, and M.L. Lee, editors, *Proc. of the 17th Int. Conference on Conceptual Modeling (ER'98), Singapore* , pages 183–196. Springer, LNCS 1507, November 1998.

[PR95] O. Pastor and I. Ramos. OASIS 2.1.1: A Class Definition Language to Model Information Systems Using an Object-Oriented Approach. Technical report, Universidad Politécnica de Valencia, March 1995.

[PS83] H. Partsch and R. Steinbrüggen. Program Transformation Systems. *ACM Computing Surveys*, 15(3), September 1983.

[Ram98] R. Ramakrishnan. *Database Management Systems*. McGraw-Hill, 1998.

[RB97] J. Robie and D. Bartels. A Comparison between Relational and Object-Oriented Databases for Object Oriented Application Development. White Paper, POET Software Corp., 1997. Available at http://www.poet.com/.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object–Oriented Modeling and Design*. Prentice Hall, New York, 1991.

[RC92] C. Rolland and C. Cauvet. Trends and Perspectives in Conceptual Modelling. In P. Loucopoulos and R. Zicari, editors, *Conceptual Modeling, Databases and CASE*, chapter 1, pages 27–41. John Wiley and Sons, Inc., 1992.

[RG97a] M. Richters and M. Gogolla. A Web-based Animator for Validating Object Specifications. In B. C. Desai and B. Eaglestone, editors, *IDEAS'97: International Database Engineering & Applications Symposium, Montreal, Canada*, pages 211–219, August 1997.

[RG97b] M. Richters and M. Gogolla. A Web-based Animator for Validating Object Specifications in a Persistent Environment. In B. C. Desai and B. Eaglestone, editors, *TAPSOFT'97: Theory and Practice of Software Development: 7th International Joint Conference CAAP/FASE, Lille, France*, pages 867–870, April 1997.

[Rom85] G.-C. Roman. A Taxonomy of Current Issues in Requirements Engineering. *Computer*, pages 14–22, April 1985.

[Roy70] W. W. Royce. Managing the Development of Large Software Systems. In *Proc. of IEEE WESCON 1970*, pages 1–9. IEEE, August 1970.

[RP92] C. Rolland and C. Proix. A Natural Language Approach for Requirements Engineering. In P. Loucopoulos, editor, *Proc. 4th Int. Conf. on Advanced Information Systems Engineering (CAiSE'92)*, pages 257–277. Springer, Berlin, LNCS 593, 1992.

[RPS95] A. Ruiz-Delgado, D. Pitt, and C. Smythe. A Review of Object-Oriented Approaches in Formal Methods. *The Computer Journal*, 38(10):777–784, 1995.

[Rüt99] T. Rütters. *Entwurf und Implementierung eines semantischen Analysators von* TROLL *Spezifikationen*. Pre-Diploma Thesis, Technische Universität Braunschweig, December 1999.

[Sai97] H. Saiedian. Formal Methods in Information Systems Engineering. In R. H. Thayer and M. Dorfman, editors, *Software Requirements Engineering*, pages 336–349. IEEE Computer Society Press, 1997.

[Sán00] P. Sánchez. *Animación de Especificaciones OASIS mediante Redes de Petri Orientadas a Objeto*. PhD thesis, Departamento de

Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2000.

[Sch97] T. Schaper. *Trollspachmodus für den Emacs-Texteditor.* Prediploma Thesis, Technische Universität Braunschweig, April 1997.

[Sch99] S. Schulte. *Entwurf und Implementierung einer Datenbank- und einer Benutzerschnittstelle für die Animation von objektorientierten Spezifikationen in* TROLL. Diploma Thesis, Technische Universität Braunschweig, April 1999.

[Sch00] C. Schmidt. *Remodellierung, Neuimplementierung und Erweiterung eines Meßdatenbearbeitungssystems für druckfeste Kapselung.* Diploma Thesis, Technische Universität Braunschweig, February 2000.

[Sel93] A. H. Seltveit. An Abstraction-Based Rule Approach to Large-Scale Information Systems Development. In C. Rolland, F. Bodart, and C. Cauvet, editors, *Proc. 5th Int. Conf. on Advanced Information Systems Engineering (CAiSE'93)*, pages 328–351. Springer, Berlin, LNCS 685, 1993.

[SF91] A. T. Schreiner and G. Friedman. *Compiler bauen mit Unix.* Carl Hanser Verlag, München, 1991.

[SFSE88] A. Sernadas, J. Fiadeiro, C. Sernadas, and H.-D. Ehrich. Abstract Object Types: A Temporal Perspective. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proc. Colloq. on Temporal Logic in Specification*, pages 324–350. LNCS 398, Springer, Berlin, 1988.

[SHJE94] G. Saake, T. Hartmann, R. Jungclaus, and H.-D. Ehrich. Object-Oriented Design of Information Systems: `TROLL` Language Features. In J. Paredaens and L. Tenenbaum, editors, *Advances in Database Systems, Implementations and Applications*, pages 219–245. Springer Verlag, Wien, CISM Courses and Lectures no. 347, 1994.

[SJE92] G. Saake, R. Jungclaus, and H.-D. Ehrich. Object-Oriented Specification and Stepwise Refinement. In J. de Meer,

V. Heymer, and R. Roth, editors, *Proc. Open Distributed Processing, Berlin (D), 8.-11. Okt. 1991 (IFIP Transactions C: Communication Systems, Vol. 1)*, pages 99–121. North-Holland, 1992.

[SK97] M. Schönhoff and M. Kowsari. Specifiying the Remote Controlling of Valves in an Explosion Test Environment. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe, FME'97, 4th Intern. Symposium, Technical University Graz, Austria, 15-19 September, 1997*, pages 201–220. Springer, Berlin, LNCS 1313, 1997.

[SLR97] P. Sánchez, P. Letelier, and I. Ramos. Constructs for Prototyping Information Systems with Object Petri Nets. In *Proc. of IEEE Systems, Man and Cybernetics, (SMC'7), Orlando (USA), vol. 5*, pages 4260–4265, October 1997.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, London, 2nd edition, 1992.

[SR94] A. Sernadas and J. Ramos. The GNOME Language: Syntax, Semantics and Calculus. Tech. Report. Technical report, Instituto Superior Técnico, Lisbon, 1994.

[SSE87] A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In P.M. Stoecker and W. Kent, editors, *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, pages 107–116. VLDB Endowment Press, Saratoga (CA), 1987.

[Ste92] Axel Stein. *Implementierung eines Parsers für eine objektorientierte Spezifikationssprache für Informationssysteme*. Diploma Thesis, Technische Universität Braunschweig, December 1992.

[Sto91] A. D. Stokes. Requirements Analysis. In J. A. McDermid, editor, *Software Engineer's Reference Book*, chapter 16. Butterworth Heinemann, 1991.

[Stö93] R. Stöcker. *Implementierung der kontextsensitiven Analyse für eine objektorientierte Spezifikationssprache für Informationssys-*

teme. Diploma Thesis, Technische Universität Braunschweig, May 1993.

[Sto94] M. Stonebraker. *Object-Relational DBMSs - The Next Great Wave*. Morgan Kaufmann, San Mateo, 1994.

[Str97] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, 1997.

[TD97] R. H. Thayer and M. Dorfman, editors. *Software Requirements Engineering*. IEEE Computer Society Press, 1997.

[Ver96] Verilog. *User Manual for VENUS OMT-VDM++ Coupling Module*, December 1996.

[VHG+93] N. Vlachantonis, R. Herzig, M. Gogolla, G. Denker, S. Conrad, and H.-D. Ehrich. Towards Reliable Information Systems: The KORSO Approach. In C. Rolland, F. Bodart, and C. Cauvet, editors, *Proc. 5th Int. Conf. on Advanced Information Systems Engineering (CAiSE'93)*, pages 463–482. Springer, Berlin, LNCS 685, 1993.

[Vie97] R. Vienneau. A Review of Formal Methods. In R. H. Thayer and M. Dorfman, editors, *Software Requirements Engineering*, pages 324–335. IEEE Computer Society Press, 1997.

[Voe99] S. Voecks. *Entwurf und Implementierung eines zentralen Repositories für die Verwaltung von* TROLL*-Spezifikationen in einer CASE-Umgebung*. Diploma Thesis, Technische Universität Braunschweig, September 1999.

[Win00] C. Winter. *Remodellierung, Neuimplementierung und Erweiterung eines Meßdatenerfassungssystems für druckfeste Kapselung*. Diploma Thesis, Technische Universität Braunschweig, February 2000.

[WJH+93] R. Wieringa, R. Jungclaus, P. Hartel, T. Hartmann, and G. Saake. OMTROLL – Object Modeling in TROLL. In U.W. Lipeck and G. Koschorreck, editors, *Proc. Intern. Workshop on Information Systems – Correctness and Reusability IS-CORE*

'93, Technical Report, University of Hannover No. 01/93, pages 267–283, 1993.

[Zav84] P. Zave. The Operational versus the Conventional Approach to Software Development. *Communications of the ACM*, February 1984.

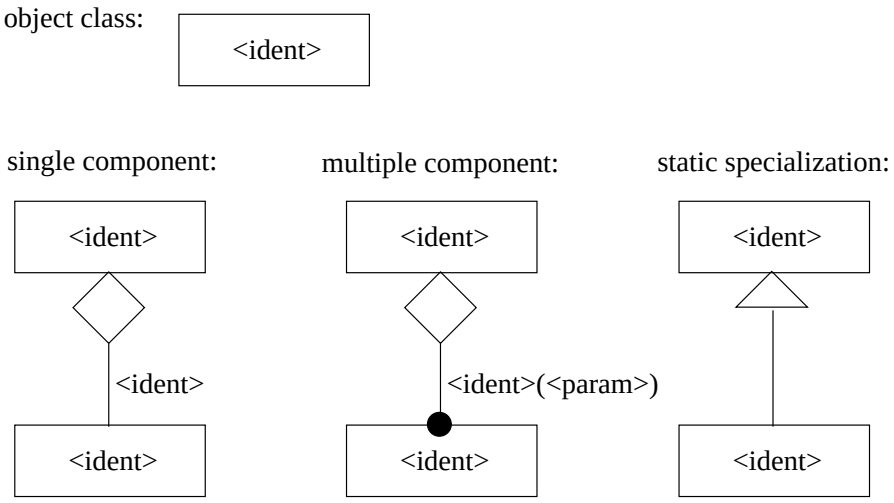
Appendix A

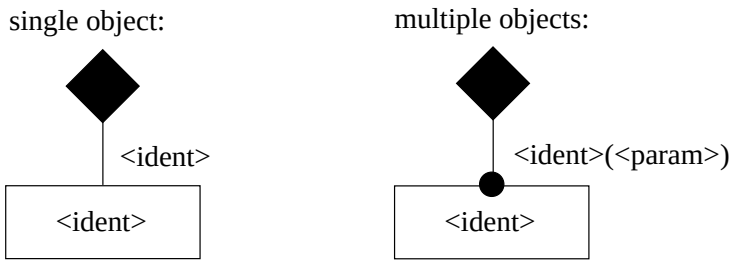
Syntax

This appendix contains all OmTroll diagrams and the Troll syntax.

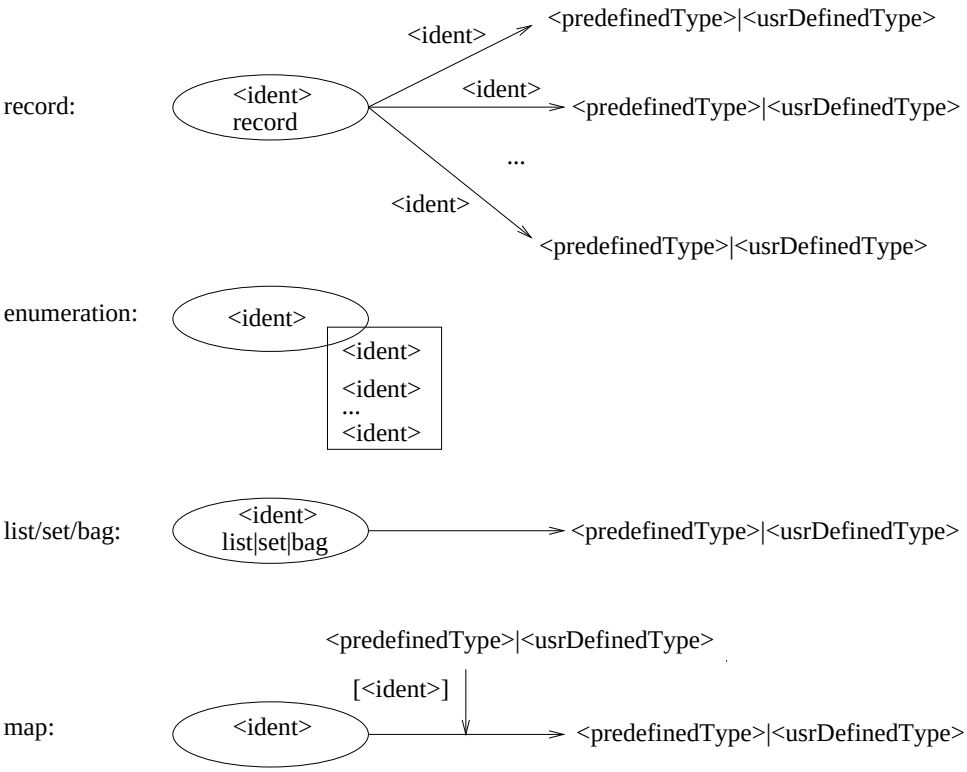
A.1 OmTroll Diagrams

Community Diagram

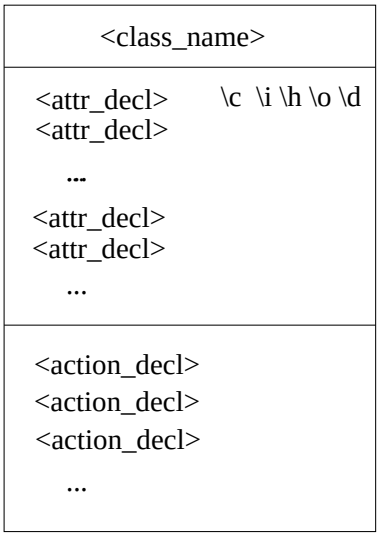




Data Type Diagram



Object Class Declaration Diagram

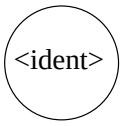


Object Behavior Diagram

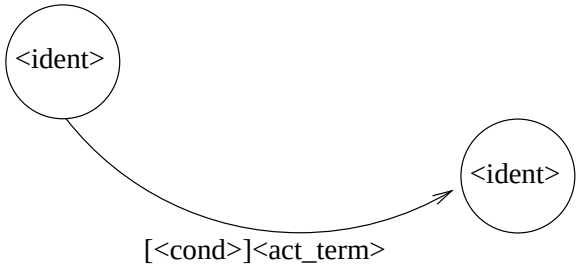
initial state:

final state:

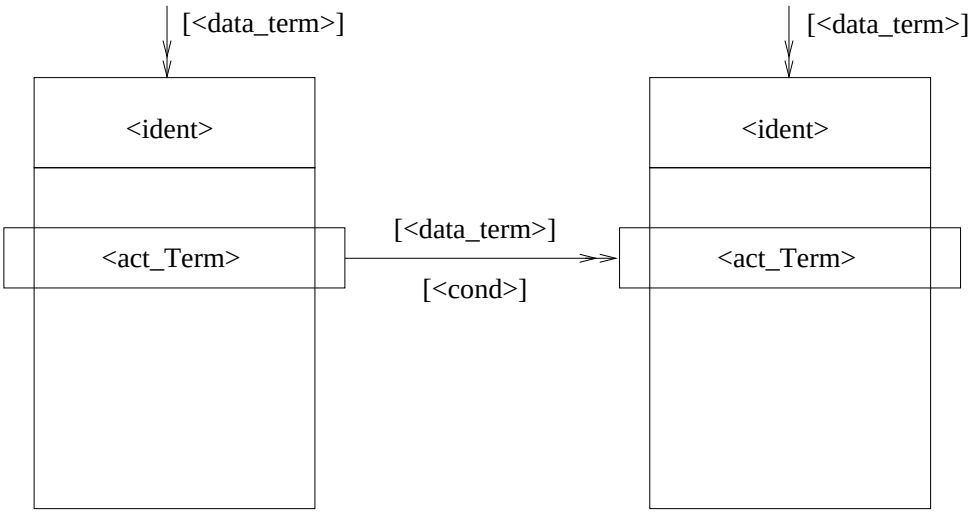
state:



transition:



Communication Diagram



A.2 TROLL Syntax

The syntax is given in EBNF format:

- A vertical line (|) separates alternatives.
- Brackets ([]) surround optional items.
- Braces ({}) surround items that can repeat zero or more times.
- Terminal symbols are written in bold face (e. g. **actions**) or are inside single-quotes (e. g. '=').

Keywords

actions	**all**	**and**	**any**	**aspect of**
attributes	**bag**	**behavior**	**bool**	**char**
cnt	**components**	**constant**	**constraints**	**data type**
date	**def**	**derived**	**div**	**do**
dom	**elem**	**else**	**empty**	**end**
enum	**false**	**fi**	**foreach**	**from**
head	**hidden**	**if**	**implies**	**in**
initialized	**initially**	**int**	**isA**	**list**
map	**mk-**	**mod**	**money**	**nat**
not	**num**	**object class**	**objects**	**od**
onlyIf	**optional**	**or**	**real**	**record**
rng	**select**	**set**	**string**	**sublist**
tail	**then**	**toSet**	**true**	**var**
where	**xor**			

Comments

Comment ::= /* ... */

LineComment ::= // ...

Identifiers and Constants

<letter> ::= 'A' | ... | 'Z' | 'Ä' | 'Ö' | 'Ü' |
'a' | ... | 'z' | 'ä' | 'ö' | 'ü' | 'ß'.

<number> ::= '0' | ... | '9'.

<ident> ::= <letter> {<letter> | <number> | '_'} .

<charConst> ::= ‘ ’ ’ asciichar ‘ ’ ’.

<stringConst> ::= ‘ ” ’ {asciichar} ‘ ” ’.

<natConst> ::= <number> {<number>} .

<intConst> ::= [‘-’] <natConst>.

<moneyConst> ::= <natConst> ‘.’ <natConst> [‘E’ [‘+’ | ‘-’] <natConst>].

<realConst> ::= [‘-’] <moneyConst>.

<boolConst> ::= **true** | **false**.

<dateConst> ::= ‘[’ <natConst> ‘,’ <natConst> ‘,’ <natConst> ‘]’.

Data Types

<type> ::= <ident> | ‘|’ <ident> ‘|’ |
enum ‘(’ <ident> {‘,’ <ident>} ‘)’ |
set ‘(’ <type> ‘)’ |
list ‘(’ <type> ‘)’ |
bag ‘(’ <type> ‘)’ |
record ‘(’ <field> { ‘,’ <field>} ‘)’ |
record ‘(’ <field> { ‘;’ <field>} ‘)’ |
map ‘(’ <field> ‘,’ <type> ‘)’ |
bool | **char** | **date** | **int** |
nat | **money** | **real** | **string**.

<dataTypeSpec> ::= **data type** <ident> ‘=’ <type>.

<field> ::= [<ident> ‘:’] <type>.

Variables

<variableDecl> ::= **var** <variable> {‘,’ <variable>} |

<variable> ::= <ident> ‘:’ <type>.

Data Terms

<constTerm> ::= <ident> |<charConst> | <stringConst> |
<natConst> |<intConst> |<moneyConst> |
<realConst> |<boolConst> | <dateConst> | **empty**.

<mapLet> ::= ‘(’ <dataTerm> ‘,’ <dataTerm> ‘)’.

<constructor> ::= **mk-list** '(' <dataTerm> {',' <dataTerm>} ')' |
mk-set '(' <dataTerm> {',' <dataTerm>} ')' |
mk-bag '(' <dataTerm> {',' <dataTerm>} ')' |
mk-record '(' <dataTerm> {',' <dataTerm>} ')' |
mk-<ident> '(' <dataTerm> {',' <dataTerm>} ')' |
mk-map '(' <mapLet> {',' <mapLet>} ')' |
mk-<ident> '(' <mapLet> {',' <mapLet>} ')'.

<relation> ::= '<' | '<=' | '=' | '# ' | '>=' | '>' | **in**.

<boolOp> ::= **and** | **implies** | **or** | **xor**.

<infixOp> ::= '+' | '-' | '*' | '/' | **div** | **isA** | **mod** |
<relation> | <boolOp>.

<prefixOp1> ::= '-' | **head** | **tail** | **cnt** | **toSet** | **rng** | **dom**.

<prefixOp2> ::= **def** | **num** | **elem**.

<condTerm> ::= '[' <formula> ? <dataTerm> ':' <dataTerm> ']'.

<selectTerm> ::= **select** <dataTerm> {',' <dataTerm>}
from '(' <rangeDecl> {',' <rangeDecl>} ')'
[**where** <formula>].

<dataTerm> ::= <ident> | <constTerm> | '(' <dataTerm> ')' |
<qualidentTerm> '.' <ident> ['(' <dataTerm> ')'] |
<qualidentTerm> '.' '@' <natConst> |
<prefixOp1> '(' <dataTerm> ')' |
<prefixOp2> '(' <dataTerm> ',' <dataTerm> ')' |
sublist '('<dataTerm> ',' <dataTerm> '..'
<dataTerm> ')' |
<dataTerm> <infixOp> <dataTerm> |
<constructor> | <condTerm> | <selectTerm>.

Qualified Identifier Terms

<qualidentTerm> ::= [<qualidentTerm> '.'] <qualidentItem> |
elem '(' <dataTerm> ',' <dataTerm> ')'.

<qualidentItem> ::= <ident> ['(' <dataTerm> ')'] | '@' <natConst>.

Propositions

<formula> ::= **not** <formula> | <formula> <boolOp> <formula> |

‘(’ <formula> ‘)’ | <dataTerm> |
all ‘(’ <rangeDecl> {‘,’ <rangeDecl>} ‘)’
‘(’ <formula> ‘)’ |
any ‘(’ <rangeDecl> {‘,’ <rangeDecl>} ‘)’
‘(’ <formula> ‘)’.

<rangeDecl> ::= <ident> **in** <dataTerm>.

Object Classes

<objClassSpec> ::= **object class** <ident>
[<specialization>]
[**components**
<componentDecl> {‘;’ <componentDecl> } ‘;’]
[**attributes**
<attribDecl> {‘;’ <attribDecl> } ‘;’]
[**actions**
<actionDecl> {‘;’ <actionDecl> } ‘;’]
[**behavior**
<operationDef> {‘;’ <operationDef> } ‘;’]
[**constraints**
<constraintRule> {‘,’ <constraintRule> } ‘;’]
end.

Signature Declaration

<specialization> ::= **aspect of** <ident>
if <specCondition> {‘,’ <specCondition>}.

<specCondition> ::= <specAction> [**and** <formula>].

<specAction> ::= <ident> [‘(’ <ident> {‘,’ <ident>} ‘)’].

<attribDecl> ::= <variable> [<attribDesc> {‘,’ <attribDesc>}].

<attribDesc> ::= **hidden** | **constant** | **optional** |
derived <dataTerm> | **initialized** <constTerm>.

<componentDecl> ::= <ident> [‘(’ <field> ‘)’] ‘:’ <ident> [**hidden**].

<actionDecl> ::= [‘+’ | ‘*’] <ident>
[‘(’ <parameter> {‘ ,’<parameter>}‘)’] [**hidden**]

<parameter> ::= [‘!’] <field>.

Behaviour Definition

<operationDef> ::= <actionDef> [**onlyIf** <formula>]
[<variableDecl>]
[**do** [<actionRule> {‘,’ <actionRule>}] **od**].

<actionDef> ::= {<qualident> ‘.’} <ident>
[‘(’ <ident> {‘,’ <ident>} ‘)’].

<qualident> ::= <ident> [‘(’ <qualident> ‘)’].

<actionRule> ::= <valuation> | <callTerm> |
<repetitiveRule> | <conditionalRule>.

<valuation> ::= <assigTerm> ‘:=’ <dataTerm>.

<assigTerm> ::= [<qualidentTerm> ‘.’] <ident> |
<qualidentTerm> ‘.’ ‘@’ <natConst> |
elem ‘(’ <dataTerm> ‘,’ <dataTerm> ‘)’.

<callTerm> ::= [<qualidentTerm> ‘.’] <ident>
[‘(’ <dataTerm> {‘,’ <dataTerm>} ‘)’].

<conditionalRule> ::= **if** <formula> **then** <actionRule> {‘,’ <actionRule>}
[**else** <actionRule> {‘,’ <actionRule>}] **fi**.

<repetitiveRule> ::= **foreach** ‘(’ <rangleDecl> ‘)’
do [<actionRule> {‘,’ <actionRule>}] **od**.

<constraintRule> ::= <formula> | **initially** <formula>.

System Specification

<systemSpec> ::= <specItem> {‘;’ <specItem>}.

<specItem> ::= <dataTypeSpec> | <objClassSpec> |
<instanceDecl> | <behaviorSpec>.

<instanceDecl> ::= **objects** <ident> [‘(’ <field> ‘)’] ‘:’ <ident>.

<behaviorSpec> ::= **behavior** <operationDef>
{‘;’ <operationDef>} ‘;’ **end**.

Appendix B

TROLL Example

This appendix contains the TROLL specification of the CATC example introduced in Chapter 3.

```
/* Data type definitions */
data type labours = enum(l3_41, l3_42, l3_43);
data type msset = record(press:real, time:real);
data type msresults = list(real);
data type address_type = record(street:string,nr:nat,city:string);
data type users_type = enum(admin, staff, operator);

/* Object class Group */
object class Group
  components
    Applications(appNr:nat) : Application;
    Companies(compId:nat) : Company;
  actions
    *create;
    +delete;
end;

/* Object class Application */
object class Application
  components
    Experiments(expNr:nat) : Experiment;
  attributes
    company : |Company| constant;
```

```
    labour : labours;
    max_pressure : real constant;
    nextExpNr : nat initialized 1, hidden;
  actions
    *createAppl(comp:|Company|,max_press:real,lab:labours);
    newExperiment(nam:string,st:msset,!expNr:nat);
    +deleteAppl;
  behavior
    createAppl(comp,max_press,lab)
    do
      company := comp,
      max_pressure := max_press,
      labour := lab
    od;
    newExperiment(nam,st,expNr)
    onlyIf(st.press <= max_pressure)
    do
      Experiments(nextExpNr).createExp(nam,st),
      expNr := nextExpNr,
      nextExpNr := nextExpNr+1
    od;
  constraints
    nextExpNr <= 11;
end;

 /* Object class Experiment */
object class Experiment
  attributes
    name : string constant;
    setup : msset constant;
    results : msresults initialized empty;
    assessments : list(string) initialized empty;
  actions
    *createExp(nam:string,st:msset);
    giveSetup(!st:msset);
    storeResults(res:msresults);
    giveResults(!res:msresults);
    storeAssessments(assmt:list(string));
    deleteExp;
  behavior
```

```
    createExp(nam,st)
    do
      name := nam,
      setup := st
    od;
    giveSetup(st)
    do
      st := setup
    od;
    storeResults(res)
    do
      results := res
    od;
    giveResults(res)
    do
      res := results
    od;
    storeAssessments(assmt)
    do
      assessments := assmt
    od;
end;

/* Object class Company */
object class Company
  attributes
    name : string;
    address : address_type;
    phone : string;
  actions
    *create(nam:string,addr:address_type,phon:string);
    +delete;
  behavior
    create(nam,addr,phon)
    do
      name := nam,
      address := addr,
      phone := phon
    od;
end;
```

```
/* Object class User */
object class User
  attributes
    shortName : string constant;
    login_date : date constant;
  actions
    *login(n:string,d:date,user_t:users_type);
    +logout;
  behavior
    login(n,d,user_t)
    do
      shortName := n,
      login_date := d
    od;
end;

/* Object class Staff */
object class Staff
  aspect of User if login(n,d,t) and t = staff
  actions
    createExperiment(appNr:nat,nam:string,st:msset,!expNr:nat);
    askResults(appNr:nat,expNr:nat,!res:msresults);
    checkResults(appNr:nat,expNr:nat,res:msresults);
    giveAssessment(appNr:nat,expNr:int,assmt:list(string));
end;

/* Object class Operator */
object class Operator
  aspect of User if login(n,d,t) and t = operator
  actions
    askSetup(appNr:nat,expNr:nat,!st:msset);
    startExperiment(appNr:nat,expNr:nat,st:msset);
    giveResults(appNr:nat,expNr:int,res:msresults);
end;

/* Object declarations */
objects IG34:Group;
objects Users(userId:nat):User;
```

```
 /* Global behavior specification */
behavior
  Staff(Users(userId)).createExperiment(appNr,nam,st,expNr)
  do
    IG34.Applications(appNr).newExperiment(nam,st,expNr)
  od;
  Staff(Users(userId)).askResults(appNr,expNr,res)
  do
    IG34.Applications(appNr).Experiments(expNr).giveResults(res)
  od;
  Staff(Users(userId)).giveAssessment(appNr,expNr,assmt)
  do
    IG34.Applications(appNr).Experiments(expNr) \
    .storeAssessments(assmt)
  od;
  Operator(Users(userId)).askSetup(appNr,expNr,st)
  do
    IG34.Applications(appNr).Experiments(expNr).giveSetup(st)
  od;
  Operator(Users(userId)).giveResults(appNr,expNr,res)
  do
    IG34.Applications(appNr).Experiments(expNr).storeResults(res)
  od;
end;
```

Appendix C

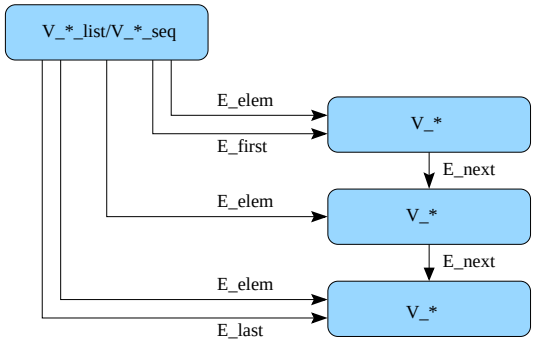
Syntax Graph

This appendix describes the structure of the syntax graph generated from the specifications by the syntax and semantic analysers. We do this by showing the representation of each element of the TROLL syntax in the graph. The structure of the graph was already introduced in Sect. 4.2. In the next, we adopt the following conventions:

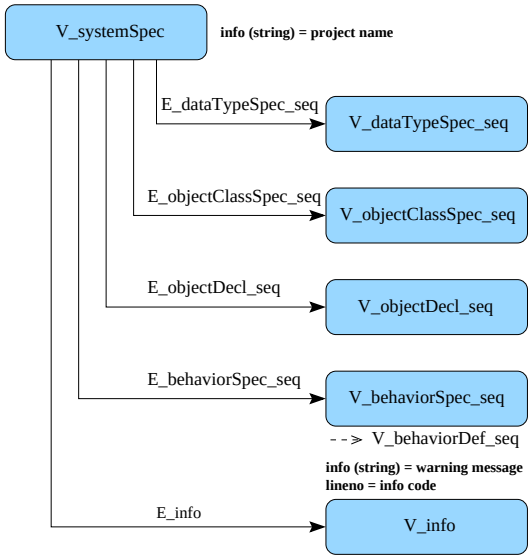
- Optional edges and vertexes are indicated by an 'o' after the labels
- Edges and vertexes added by the semantic analyser are followed by a plus character ('+')
- Edges and vertexes removed by the semantic analyser are followed by a minus character ('-')
- Vertexes in parentheses do not appear in the graph. They are substituted with other vertexes.
- Vertexes whose labels end in '`_id`' represent string identifiers.

Lists and Sequences

To simplify the presentation of the graph, we do not show the representation of each possible list or sequence, but we summarise them in the next figure where "*" must be substituted with the corresponding label.

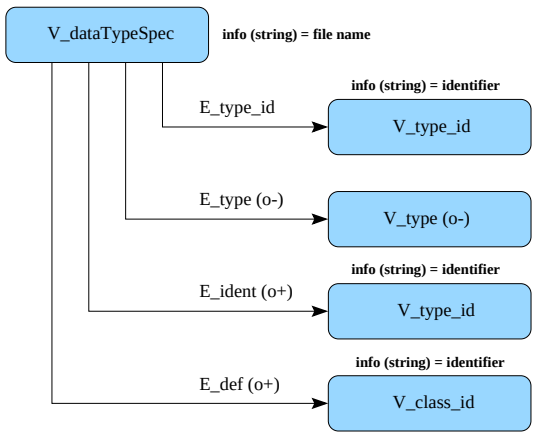


System Specification



Elements of `V_behaviorSpec_seq` are sequences of behaviour definitions (`V_behaviorDef_seq`), which are shown later in the appendix. `V_info` contains status information required in the subsequent phases such as whether code can be generated from the specification or not.

Data Type Specification



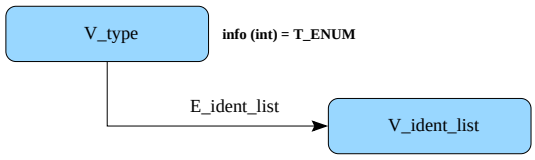
The edge `E_type` and its target vertex `V_type` are removed by the semantic analyser if the specified user data type is either an object reference or another user defined data type. In the former, an edge `E_def` pointing to the corresponding class declaration is added. In the latter, the corresponding data type specification is pointed to by a new edge `E_ident`.

Data Types

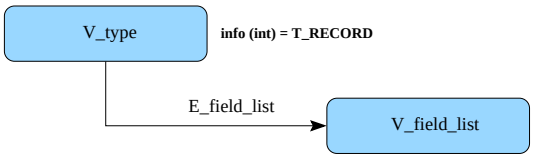
- Simple Types

If the data type is a simple predefined data type, the vertex `V_type` is removed and substituted by one of the following vertexes:

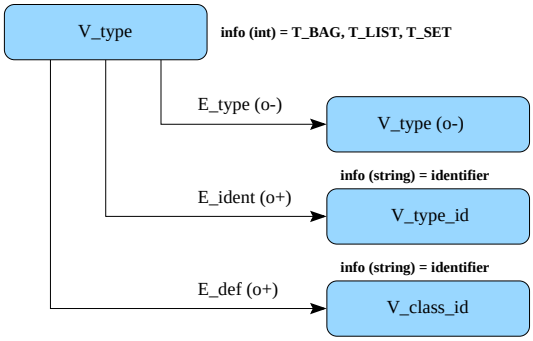
- Enumerations



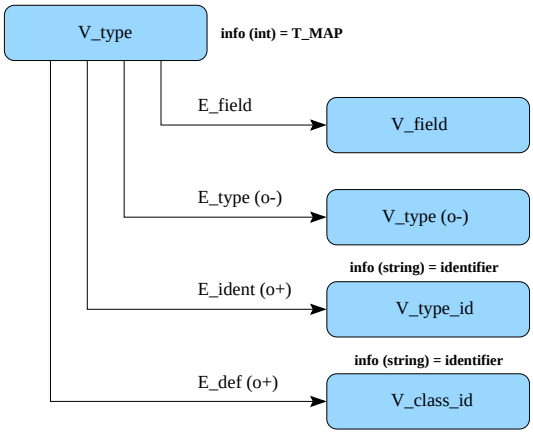
- Records



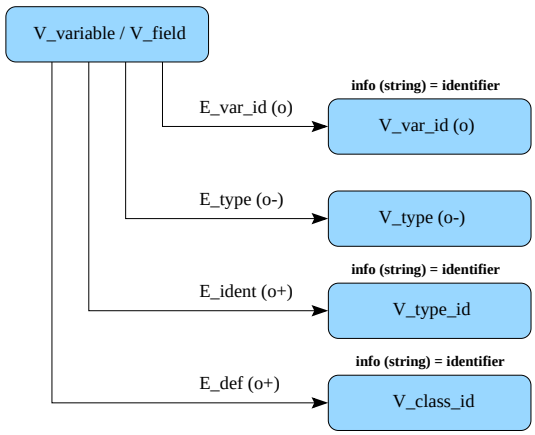
- Lists, Sets and Bags



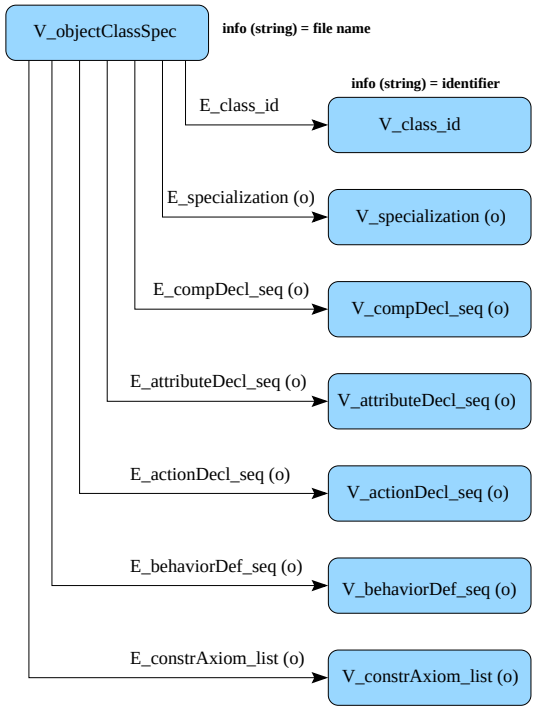
- Maps



Variables and Fields

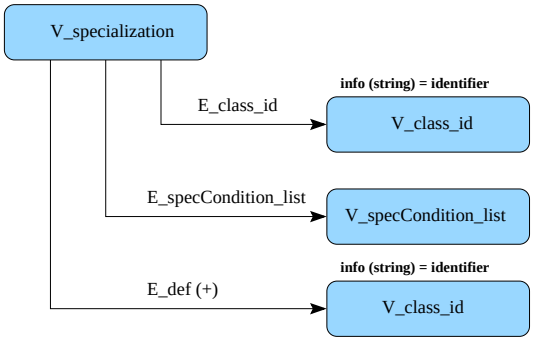


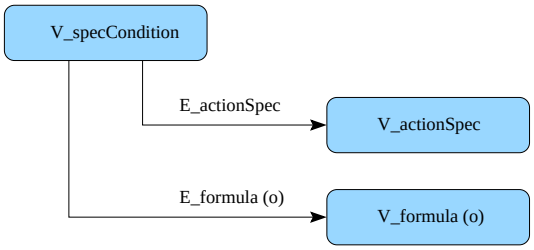
Object Class Specification

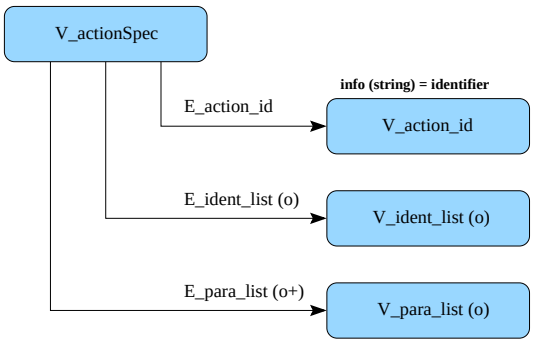


The edge `E_actionDecl_seq` and its target vertex `V_actionDecl_seq` are only optional if the class is a specialisation. If not, at least a birth action must be declared.

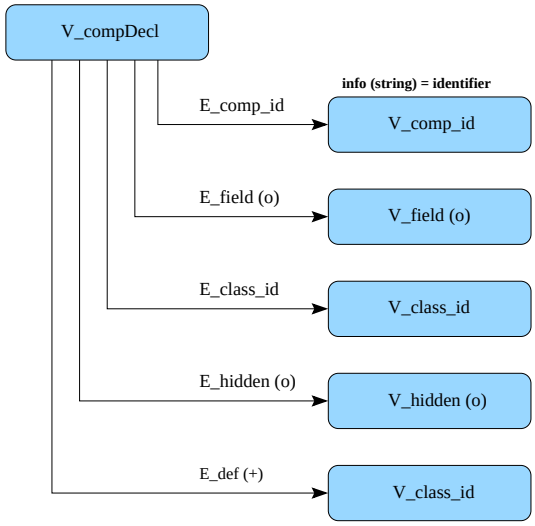
Specialisations



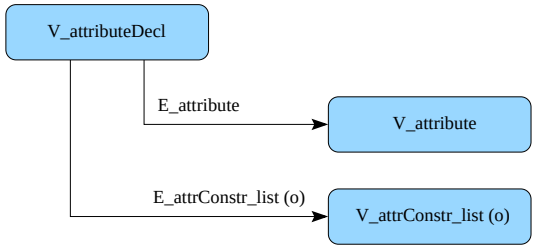


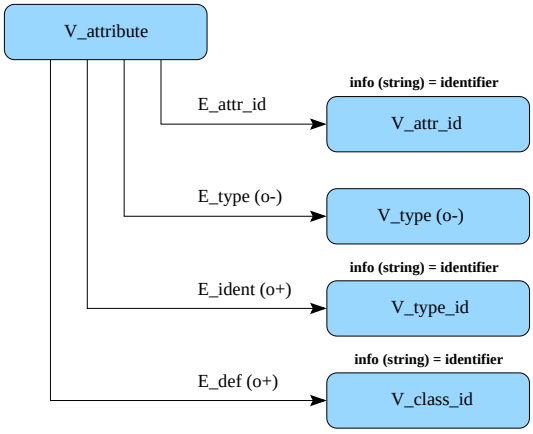


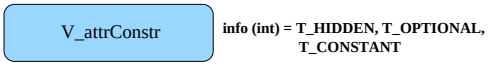
Component Declaration

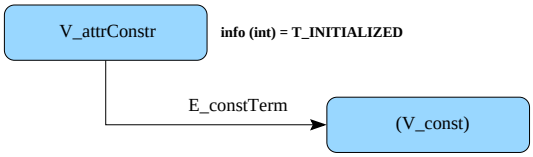


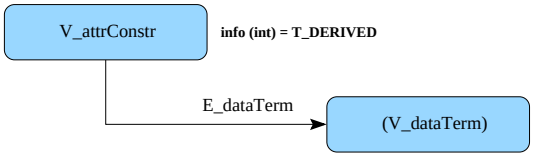
Attribute Declaration



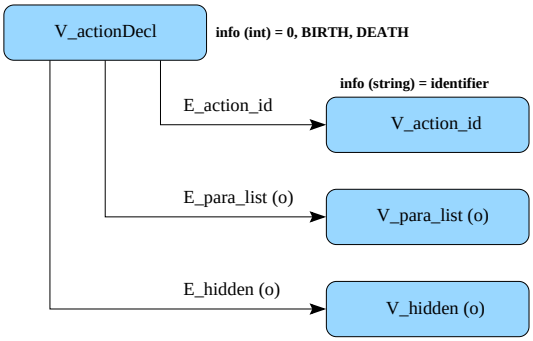

V_attribute
E_attr_id
info (string) = identifier
V_attr_id
E_type (o-)
V_type (o-)
E_ident (o+)
info (string) = identifier
V_type_id
E_def (o+)
info (string) = identifier
V_class_id

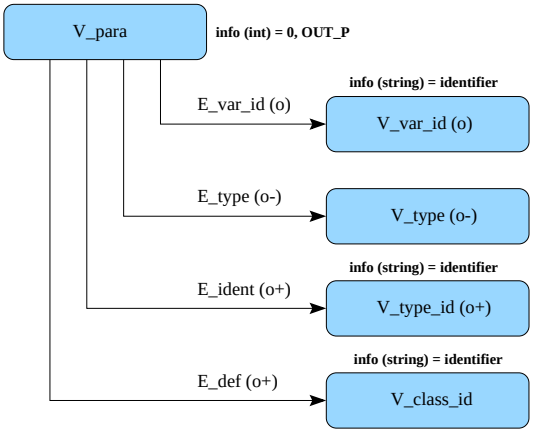

V_attrConstr
info (int) = T_HIDDEN, T_OPTIONAL, T_CONSTANT


V_attrConstr
info (int) = T_INITIALIZED
E_constTerm
(V_const)

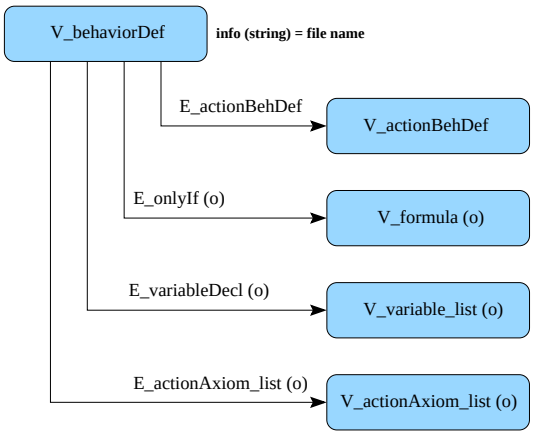

V_attrConstr
info (int) = T_DERIVED
E_dataTerm
(V_dataTerm)

Action Declaration

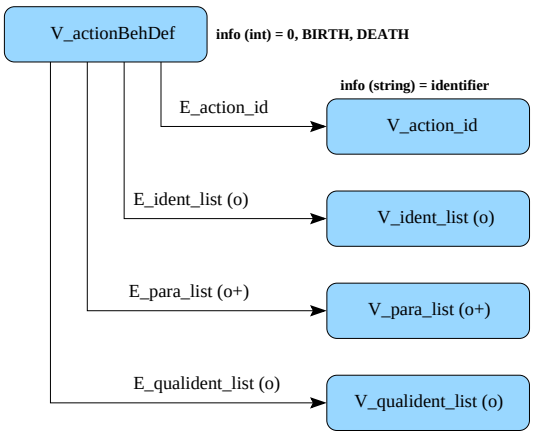


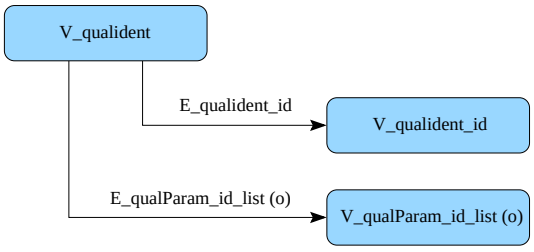


Behaviour Definition



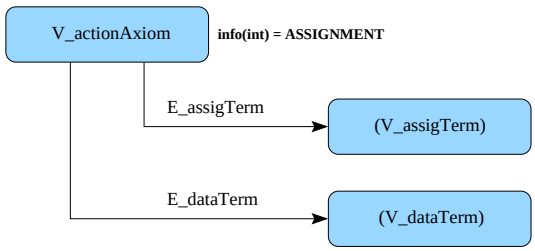
Action Definition



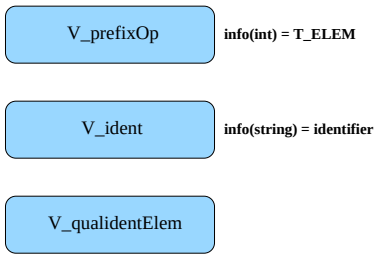


Action Rules

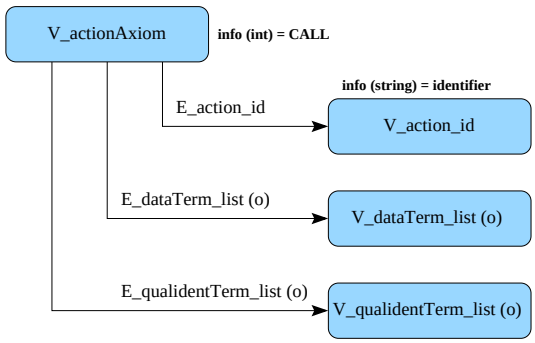
- Assignments



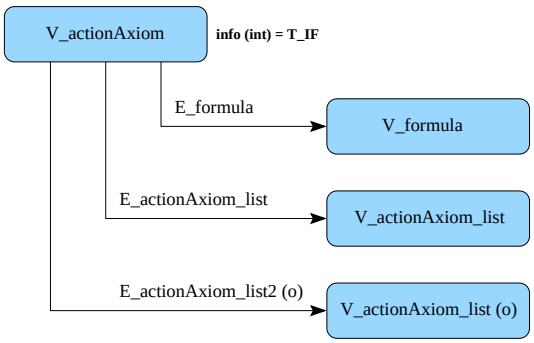
Vertexes `V_assigTerm` are removed from the graph and substituted by the following ones:



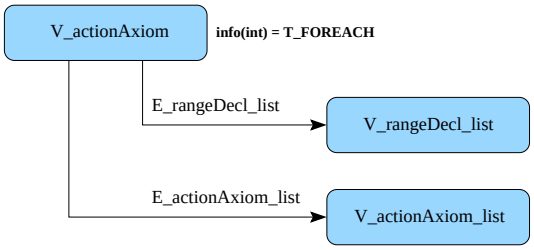
- Action Calling



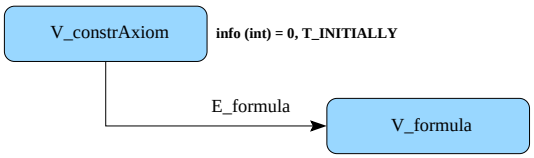
- Conditionals



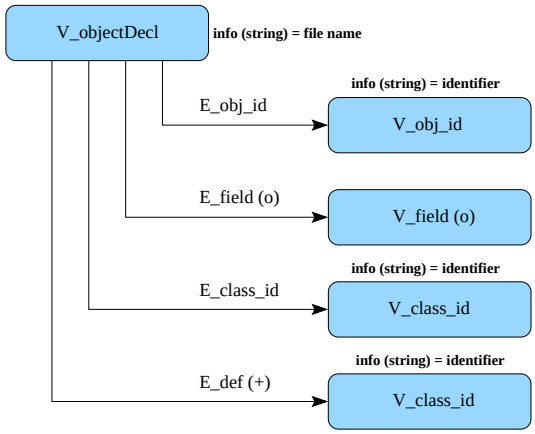
- Iterations



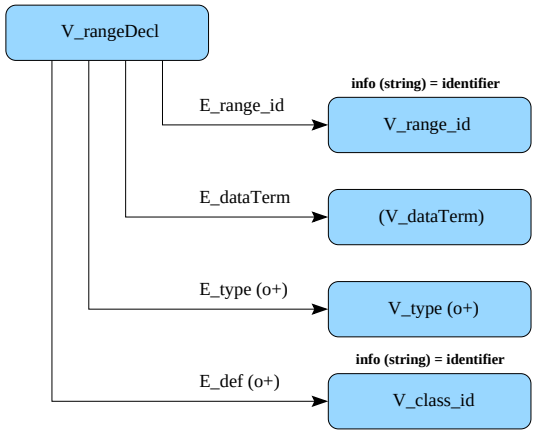
Constraint Declaration



Object Declaration

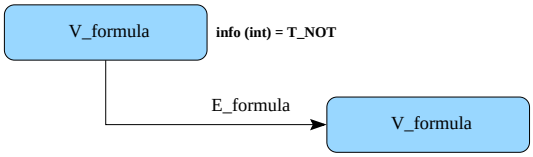


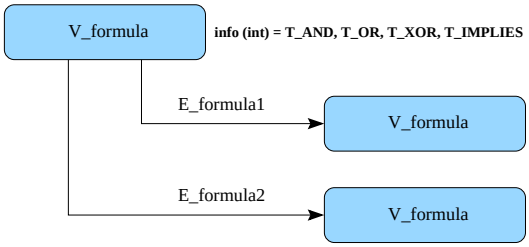
Range Declaration

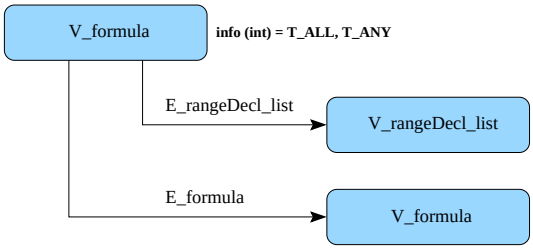


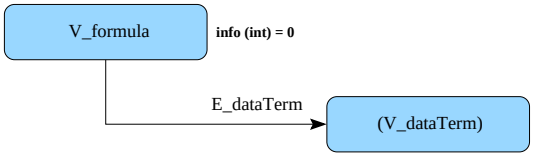
The vertex `V_dataTerm` always represents a data term of type *set*. The vertex `V_type` indicates the type of the set elements. If the type is an object reference, the edge `E_def` points to the corresponding class declaration.

Formulas


V_formula
info (int) = T_NOT
E_formula
V_formula


V_formula
info (int) = T_AND, T_OR, T_XOR, T_IMPLIES
E_formula1
V_formula
E_formula2
V_formula


V_formula
info (int) = T_ALL, T_ANY
E_rangeDecl_list
V_rangeDecl_list
E_formula
V_formula

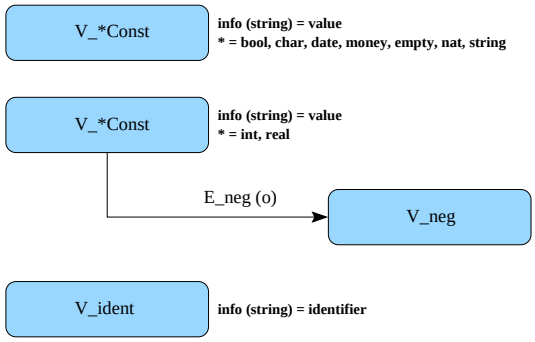

V_formula
info (int) = 0
E_dataTerm
(V_dataTerm)

Data Terms

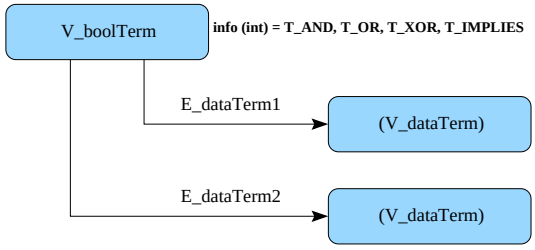
Vertexes `V_dataTerm` are removed from the graph and substituted by the following ones.

- Constant Terms

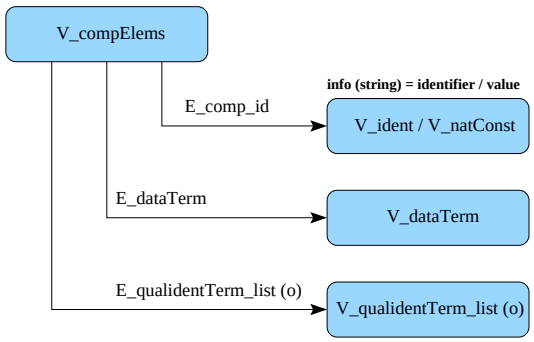
Vertexes `V_const` are substituted by the next ones:



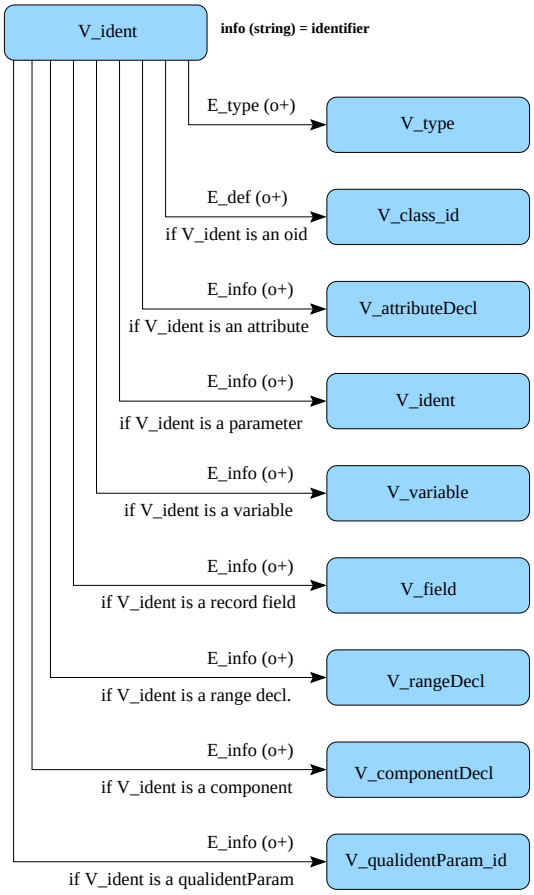
- Boolean Terms



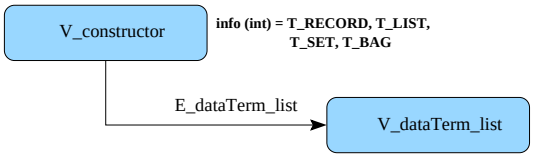
- Component Terms

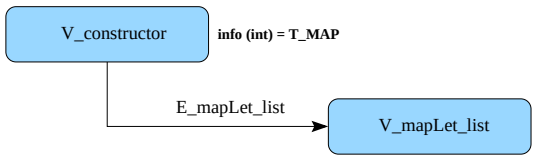


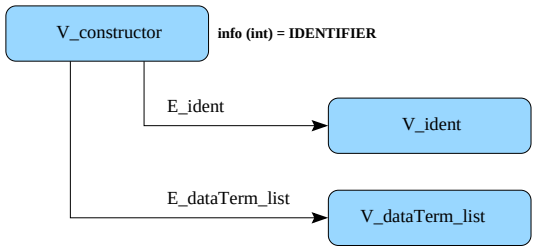
- Identifiers

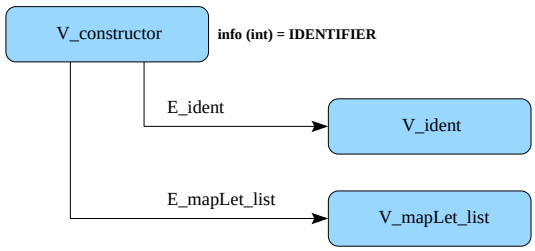


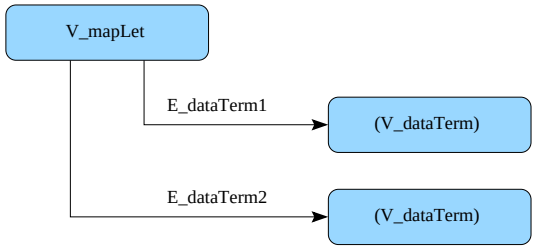
- Constructors



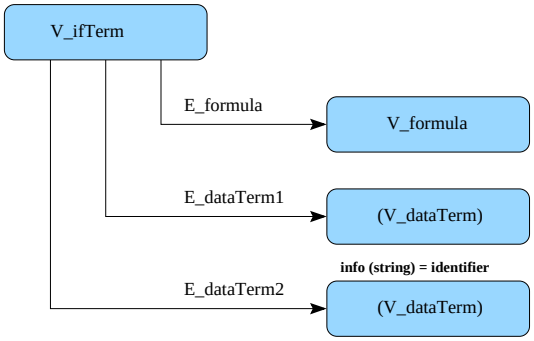

V_constructor
info (int) = T_MAP
E_mapLet_list
V_mapLet_list


V_constructor
info (int) = IDENTIFIER
E_ident
V_ident
E_dataTerm_list
V_dataTerm_list

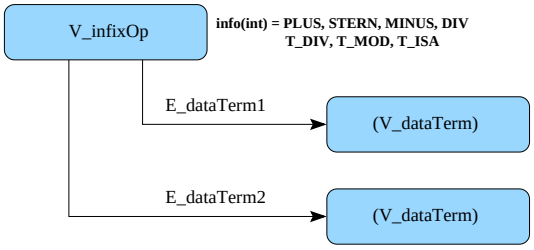

V_constructor
info (int) = IDENTIFIER
E_ident
V_ident
E_mapLet_list
V_mapLet_list


V_mapLet
E_dataTerm1
(V_dataTerm)
E_dataTerm2
(V_dataTerm)

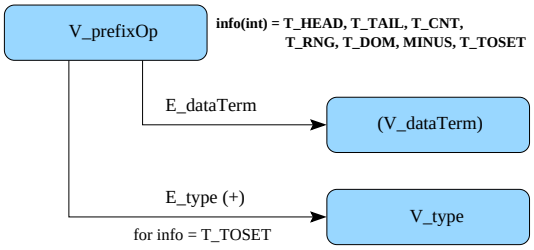
- Conditional Terms

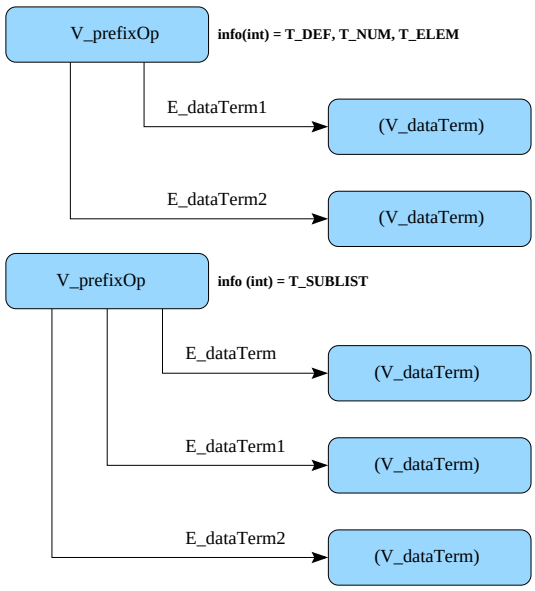


- Infix Operators

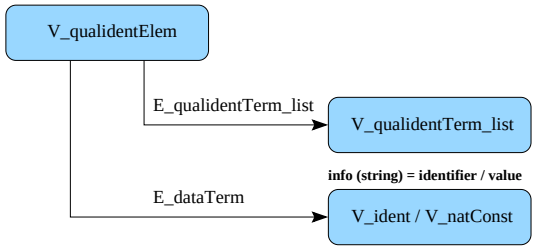


- Prefix Operators

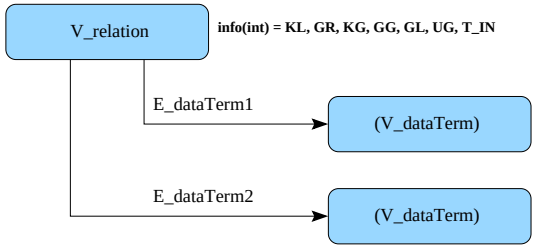




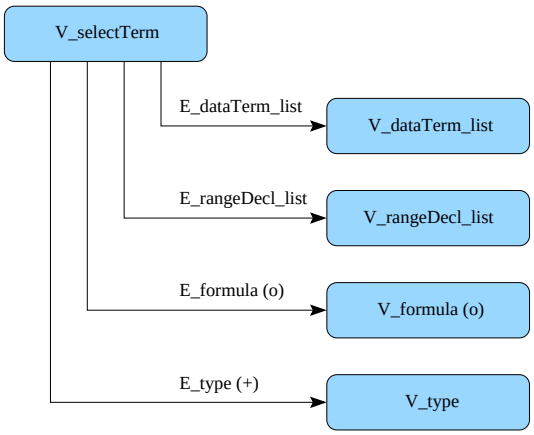
- Qualident Elements



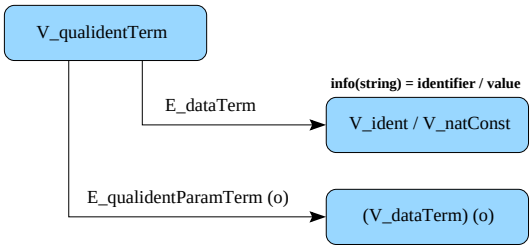
- Relation Operators



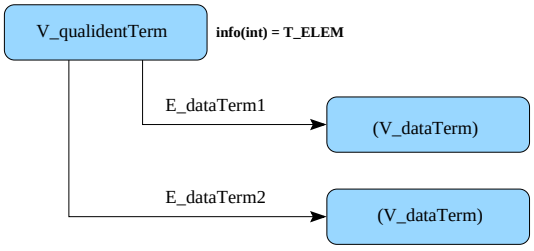
- Select Terms



Qualident Terms



The first element of a qualident term may also be a list or map element:



Appendix D

Transformation from TROLL Specifications into C++

This appendix describes the translation of TROLL specifications into C++ code which is executed for the animation. The first section describes the generic translation rules and the structure of the C++ code. The second section shows an extract of the code generated from the CATC example.

D.1 Code Generation

The following conventions will be used in the description of the code:

- Comments about the generated code are preceded by a ⇒
- Literal C++ code is written in a sans serif font.
- Three dots (...) in the code indicates that it may be repeated several times.
- Words written in an *emphasised Roman* font are substituted by code which depends on the specification elements (e. g. *action_name* must be substituted by the name of the action in the TROLL specification), or by code given in other translation rules (e. g. *formula* must be substituted by code described in the translation of formulas).

D.1.1 Data Types

Every TROLL predefined data type and constructor with the exception of records has been implemented in a C++ class. Data type classes were already introduced in Chapter 6 on page 120. Besides the implementation of each type operator, a data type class contains code to check that no variable or attribute is used without having a value and that it is assigned a unique value. User-defined data types do not explicitly appear in the code. They are translated into their basic types. The next table lists the representation of data types in the code. Object-valued types are implemented by a class

TROLL **Data Type**	**C++ Data Type**
nat	tnat
int	tint
real	treal
money	tmoney
char	tchar
string	tstring
bool	tbool
date	tdate
$\|class\_name\|$	oid<*class_name*>
enum	tstring
record	*record_name*_rec
list(*type*)	tlist<*type*>
set(*type*)	tset<*type*>
bag(*type*)	tbag<*type*>
map($type_1$,$type_2$)	tmap<$type_1$,$type_2$>

Table D.1: Data Types

template which contains a reference to an object of the corresponding class. Enumerations are represented by strings. Collection types are implemented using the container types of the C++ standard library. A C++ class is generated for each record type as follows.

⇒ Class head

```
class record_name_rec {
  public:
```

⇒ For each record field an attribute declaration is generated. If a field does not have a name, the attribute name is generated automatically.

```
  field_type field_name_;
```

⇒ An equal operation is defined. Two records are equal if their fields are equal.

```
  bool operator == (const record_name_rec& r) const {
    return (field_name_ == r.field_name_ && ...);
  }
```

⇒ Inequality operation

```
  bool operator != (const record_name_rec& r) const {
    return !(*this == r);
  }
```

⇒ To allow the definition of sets of records, a comparison operation must be provided. This is necessary so a set can be iterated in a well defined order. Since we use "<" as generic comparison operator for sets, this operator must be defined. It just returns the comparison value of the first record fields.

```
  bool operator < (const record_name_rec& r) const {
    return field_name_1_ < r.field_name_1_;
  }
```

⇒ A function returns a boolean value indicating whether the record has already been assigned. The returned value is true if all record fields have a value.

```
  bool has_value() {
    return (field_name_.has_value() && ...);
  }
```

⇒ End of class definition

```
}; // record_name_rec_
```

D.1.2 Data Terms

Constant Terms

Most of TROLL constant terms are represented in the same way in the code. Exceptions are dates and enumerators which are translated into string constants. The next table shows the representation of constant terms in the code.

TROLL **Constant Term**	**C++ Constant Term**
nat_const	*nat_const*
int_const	*int_const*
real_const	*real_const*
money_const	*money_const*
char_const	*char_const*
string_const	*string_const*
bool_const	to_lower(*bool_const*)
date_const	"*date_const*"
enum_identifier	"*enum_identifier*"

Table D.2: Constant Terms

Sort Constructors

Constructors of container types are transformed in a similar way. Since containers can be created with any number of elements, the constant NULL at the end of the argument list indicates that all arguments have been read. Constructors of user-defined data types (make-*user_type_name*) are trans-

TROLL **Constructor**	**C++ Constructor**
mk-list($dt_1,\ldots,dt_n$)	mk_list($dt_1,\ldots,dt_n$,NULL)
mk-set($dt_1,\ldots,dt_n$)	mk_set($dt_1,\ldots,dt_n$,NULL)
mk-bag($dt_1,\ldots,dt_n$)	mk_bag($dt_1,\ldots,dt_n$,NULL)
mk-map(($dt_{1_1},dt_{2_1},\ldots,(dt_{1_n},dt_{2_n})$)	mk_map((make_pair(dt_{1_1},dt_{2_1}),..., make_pair((dt_{1_n},dt_{2_n}),NULL)

Table D.3: Sort Constructors

lated into constructors of their basic types. For the construction of records, fields need to be assigned explicitly. To this end, record values are first assigned to a temporary variable which is then used in the place of the make-record term. The generated code is as follows:

$\boxed{\Rightarrow}$ A temporary variable of the record type is declared.
*record_name*_rec *record_name*_tmp;

$\boxed{\Rightarrow}$ Each value of the mk-record is assigned to the corresponding record field.
*record_name*_tmp.*field_name*_ = *data_term*;

Operators

The translation of relation, infix, prefix and boolean operators into the C++ code is illustrated in the following tables.

TROLL **Relation Op.**	**Description**	**C++ Relation Op.**
dt_1 = dt_2	equal (all types)	dt_1 == dt_2
dt_1 # dt_2	not equal (all types)	dt_1 != dt_2
dt_1 > dt_2	greater than (numerics)	dt_1 > dt_2
dt_1 >= dt_2	greater than or equal (numerics)	dt_1 >= dt_2
dt_1 < dt_2	less than (numerics), proper subset of (sets)	dt_1 < dt_2
dt_1 <= dt_2	less than or equal (numerics), subset of (sets)	dt_1 <= dt_2
dt_1 in dt_2	is an element of (sets)	dt_2.in(dt_1)

Table D.4: Relation Operators

TROLL **Infix Op.**	**Description**	**C++ Infix Op.**
dt_1 + dt_2	addition (numerics), union (sets, bags and maps), concatenation (string and lists)	dt_1 + dt_2
dt_1 - dt_2	substraction (numerics), difference (sets and bags)	dt_1 - dt_2
dt_1 * dt_2	multiplication (numerics), intersection (sets)	dt_1 * dt_2
dt_1 / dt_2	division (numerics), symmetrical difference (sets), deletion (bags and maps)	dt_1 / dt_2
dt_1 div dt_2	integer division (numerics)	tdiv(dt_1,dt_2)
dt_1 mod dt_2	modulo (numerics)	tmod(dt_1,dt_2)
dt_1 isA dt_2	is a specialisation of class	dt_1.isA("dt_2")

Table D.5: Infix Operators

TROLL **Prefix Op.**	**Description**	**C++ Prefix Op.**
-	unary minus (numerics)	-
head(dt)	first element selection (lists)	head(dt)
tail(dt)	tail of a list (lists)	tail(dt)
cnt(dt)	number of elements (lists, sets, bags and maps)	cnt(dt)
toSet(dt)	convertion to set (lists and bags)	toSet(dt)
dom(dt)	domain (maps)	dom(dt)
rng(dt)	codomain (maps)	rng(dt)
def(dt_1,dt_2)	element of (maps)	def(dt_1,dt_2)
num(dt_1,dt_2)	frequency of occurrence (bags)	num(dt_1,dt_2)
elem(dt_1,dt_2)	element selection (lists and maps)	dt_1(dt_2)
sublist(dt_1,dt_2..dt_3)	sublist selection (lists)	dt_1(dt_2,dt_3)

Table D.6: Prefix Operators

TROLL **Boolean Op.**	**C++ Boolean Op.**
dt_1 or dt_2	dt_1 || dt_2
dt_1 and dt_2	dt_1 && dt_2
dt_1 implies dt_2	!dt_1 || dt_2
dt_1 xor dt_2	dt_1 && !dt_2 || !dt_1 && dt_2

Table D.7: Boolean Operators

Conditional Terms

Conditional terms ([*formula* ? *data_term* : *data_term*]) are the same in the C++ code but without surrounding brackets.

Select Terms

The result of a select term (select *data_term*, ... from (*var_name* in *set_term*, ...) [where *formula*]) is a bag of records whose fields result from the evaluations of the data terms that appear after the keyword select. These data terms are evaluated for all assignments of the variables defined after the keyword from that satisfy the where clause. In the C++ code, the select results

are assigned to a temporary variable which is then used in the place of the select term. Next, we show the C++ implementation. Select terms can be used for formulating queries about component objects (by applying the dom or rng map operators on the components). In this case and since components are stored in a database, another possible and more efficient implementation would consist in the direct use of the SQL query provided by the DBMS.

⇒ A bag is declared to hold the select results.

tbag<*record_name*_rec> *result*;

⇒ For each variable defined after the keyword from, a loop is generated. In the loop, an iterator variable (*iter*) is declared. This iterator is a pointer to elements of the set of values from which the variable is assigned. In each iteration, the value pointed by the iterator is assigned to the variable (*range_vble_name*_)

for (*set_type*::iterator *iter* = *set_term*.begin();
    iter != *set_term*.end();
    iter++) {
    *set_of_type range_vble_name*_ = **iter*;

⇒ If a where clause is given, the selected values must satisfy the formula

    if (*where_formula*) {

⇒ Values are inserted in the bag.

    *record_name*_rec *rec*;
    rec.field = *data_term*; ...
    result.insert(*rec*);

⇒ The conditional, if defined, is closed.

  }

⇒ Every loop generated previously must be closed.

}

Identifiers and Qualidents

In the code, identifiers, whith the exception of class names, are converted to lowercase and suffixed by an underscore "_" to avoid conflicts with C++ keywords. The first letter of a class name is converted to uppercase and the following letters are converted to lowercase. A TROLL class attribute is represented by two attributes in the C++ classes: one containing its value in the current state (*attr_name*_) and another one (*attr_name*_new) containing its value in the temporary new state. Attributes referenced in data terms are

translated into one of these attributes depending on the context. The access to the signature of a superclass is done through the attribute superclass which is a pointer to the superclass object. So references to components, attributes and actions of the superclass are preceded by "superclass ->". Since object-valued types are implemented by pointers, members of referred objects are selected by using the member selection operator "->". Access to record fields, components, global objects and specialisations is explained next.

- **Access to Record Fields**

In TROLL, fields of record types are selected either by their names ("." + *field_name*), if they are named, or by their positions in the field declaration (".@" + *nat_const*). In the C++ code, fields are also selected by the use of dot notation but they are always referenced by their names. So references to record fields indicating the field positions are translated into references using the respective field names. As mentioned previously, if a field has been declared without a name, a name is generated automatically. So fields may always be selected by their names.

- **Access to Components**

Components are accessed through class attributes (*comp_name_*). A single component is represented by a reference to an object of the component class. A multiple component is implemented by a map whose domain represents component identifier values and whose codomain represents references to component objects. Before the access to component members, the component must be created on main memory if it was not created previously. This is done by the functions get_comp and get_all_comps. They are called as follows:

⇒ The kind of participation of the component in the state transition (BIRTH, DEATH, UPD or READ) is passed in the function argument *exec_type*.

⇒ Single components

get_comp(*comp_name_*,"*comp_class*","*comp_name*",*exec_type*);

⇒ In case of multiple components, the parameter identifier value and its type are additionally passed in the argument list.

get_comp(*comp_name_*,*par_term*,"*comp_class*","*comp_name*","*par_type*",*exec_type*);

⇒ In some cases such as quantified formulas or for each rules, it is necessary to load all component objects on main memory. This is done by calling the next function.

get_all_comps(*comp_name_*,"*comp_class*","*comp_name*",*exec_type*);

⇒ Once objects have been loaded on main memory, they are accessed by the corresponding attribute (*comp_name_*->... for single components and *comp_name_*[*par_term*]->... for multiple components.)

- **Access to Global Objects**

The access to objects in the global behaviour definition of actions is defined as follows.

⇒ First, a variable is declared to hold a pointer to the required object.

*obj_class_name** *object*;

⇒ Similar to get_comp, the function get_global_obj creates the object on main memory if it was not created before, and assigns its memory address to the pointer variable created previously (*object*).

⇒ For single objects

get_global_obj(*object*,"*obj_class_name*","*obj_name*",*exec_type*);

⇒ For multiple objects

get_global_obj(*object*,*par_term*,"*obj_class_name*","*obj_name*","*par_type*",*exec_type*);

⇒ Actions in the object are then called through the object pointer (*object*->...).

- **Access to Specialisations**

The access to specialisations of components and objects is performed by the function get_spec.

⇒ First, a pointer to the specialisation object is declared. The variable name is automatically generated.

*spec_class_name** *spec_object*;

⇒ The function get_spec assigns a reference to the specialisation object to the variable created before (*spec_object*). This function is accessed through the global or component object (*qualident_term*) which has the specialisation.

qualident_term->get_spec(*spec_object*,"*spec_class_name*");

⇒ Specialisation members are accessed through the created variable (*spec_object*->...).

D.1.3 Formulas

Formulas are built by boolean terms. The translation of boolean operations (or, and, implies and xor) was shown on Table D.7. Additionally, formulas may be negated (not) and quantified (all and any). Negation is just translated into the C++ negation operator "!" which also precedes the formula. The truth value of formulas which are existentially or universally quantified are first computed on temporary variables which are then used in the place of the formulas. Formulas quantified universally are implemented as follows.

⇒ A boolean variable is declared and initialised true. This variable holds the truth value of the formula to be implemented.

```
bool result = true;
```

⇒ For each variable bounded by the quantifier, a loop is generated. In the loop, an iterator variable (*iter*) is declared which is a pointer to elements of the set of values from which the variable is assigned. The loop ends when the formula has been computed true for all possible values or a value has been found for which the formula does not hold. In each iteration, the value pointed by the iterator is assigned to the bounded variable (*bound_vble_name_*).

```
for (set_type::iterator iter = set_term.begin();
    iter != set_term.end() && result;
    iter++) {
    set_of_type bound_vble_name_ = *iter;
```

⇒ If the formula does not hold for a variable instance, the quantified formula becomes false.

```
    if (! formula)
      result = false;
```

⇒ Every loop generated previously must be closed.

```
}
```

Conversely, formulas quantified existentially are translated as follows.

⇒ A boolean variable is declared and initialised false

```
bool result = false;
```

⇒ As in universally quantified formulas, a loop and a iterator are generated for each bounded variable. In this case, the loop ends when a variable instance is found for which the formula holds or when the formula is not satisfied for any instance of the values set. In each iteration, the value pointed by the iterator is assigned to the bounded variable (*bound_vble_name_*).

for (*set_type*::iterator *iter* = *set_term*.begin();
iter != *set_term*.end() && !*result*;
iter++) {
*set_of_type bound_vble_name_* = **iter*;

⇒ If a variable instance is found for which the formula holds, the quantified formula becomes true.

if (*formula*)
result = true;

⇒ Every loop generated previously must be closed.
}

D.1.4 Class Definition

The C++ class generated from each TROLL class is defined as follows.

⇒ The class is derived from the class troll_class which contains common attributes and services of the classes. The object identity is passed as argument (ident) in the constructor of the class. The class constructor calls in turn the constructor of the base class which stores the object identity in an attribute.

class *class_name* : public troll_class {
public:
class_name(const identList& ident) : troll_class (ident) {};

⇒ If the class has attributes

// attributes

⇒ For each attribute in the class

*attr_type attr_name_*;
*attr_type attr_name*_new;

⇒ If the class has components

// components

⇒ For each single component

*comp_class_name** *comp_name_*;

⇒ For each multiple component

map<*param_type*,*comp_class_name**> *comp_name_*;

⇒ If the class is a specialisation

// pointer to superclass
*superclass_name** superclass;

⇒ Begin of methods declaration

// methods

⇒ For each action declared in the signature and, if the class is a specialisation, each action of the superclass whose behaviour is extended in the class, a method is declared. Action parameters are written as follows: *param_type*& *param_name_* and separated by comas. Input parameters are constant and are preceded by const.

void *action_name* (*param_type*& *param_name_*,...);

⇒ For each action of components whose behaviour is extended in the class, a method is declared. Variables used in the identification of multiple components are passed to the method by constant arguments.

void *action_name*_comp (const *param_id_type*& *param_id_name_*,...,
param_type& *param_name_*,...);

⇒ If the class has constraints

// evaluate constraints
void check_constraints(const bool& initial=false);

⇒ If the class has attributes

⇒ For each attribute, a method returns the attribute value in the new state

// return attribute value in the new state
attr_type get_*attr_name*_new();

⇒ If the class has initialised attributes

// assign initialised attributes
void init_attributes();

⇒ If the class has derived attributes

// evaluate derived attributes
void eval_der_attributes();

⇒ Interface with the database

// read attribute values from the database
void read_attributes();
// write attribute values into the database
void write_attributes();

⇒ End of class definition

}; // *class_name*

D.1.5 Behaviour Definition

This section describes the implementation of each action defined in the C++ class.

• Definition of TROLL Actions

⇒ The function head consists of the class name, action name and, if existing, the argument list. The argument list is built as defined in the previous section.

```
void
class_name::act_name_ (param_type& param_name_,...) {
```

⇒ Control Code

```
 // Control Code
```

⇒ Check if the action instance was already executed in the state transition

⇒ If no parameters are defined in the action

```
 if (in_trace(''action_name''))
  return;
```

⇒ If the action has parameters

```
 vector<local_trace>::iterator i;
 for (i = trace.begin(); i != trace.end(); i++)
  if (i ->action == ''action_name'')
```

⇒ Check if the values of the input parameters are the same. For each input parameter and let *pos* be its position in the parameter list, the function equal is called as follows.

```
   if(equal(i->params[pos],param_name) && ...) {
```

⇒ Output parameters are assigned and the function returns. Output parameters are assigned by the function assign_param(i->params[*pos*],*param_name_*), where *pos* indicates the position of the parameter in the parameter list.

```
     assign_param(i->params[pos],param_name); ...
     return;}
```

⇒ If the action is a birth action, it must be checked that neither birth nor death actions are contained in the local execution trace. Furthermore, it must be checked that the number of objects created of a class does not exceed the limit. The function check_birth checks these conditions and throws an exception if they are not fulfilled.

```
 check_birth(''action_name'');
```

⇒ If the action is a death action, the function check_death checks that neither birth nor death actions are contained in the execution trace of the object.

```
 check_death(''action_name'');
```

⇒ The function to_trace stores the action instance in the execution trace. Moreover, it checks that the number of times the action has been executed in the object has not reached the boundary. Since this function may have any number of arguments depending on the number of parameters defined in the TROLL action, a NULL constant at the end of the argument list is required.

```
int pos = to_trace("act_name", &param_name_, ..., NULL);
```

⇒ Begin of the code implementing the action behaviour defined in the TROLL class.

```
// Behaviour code
```

⇒ If a precondition exists

```
// Check precondition
if (! precondition_formula)
  throw_exception("Precondition not fulfilled: \n precondition");
```

⇒ If the action is a birth action, and the object class has initialised attributes

```
init_attributes();
```

⇒ If the action uses local variables, they are declared.

```
var_type var_name_; ...;
```

⇒ The code associated to each action rule defined in the action behaviour is written. Previous to each action rule, code to show the execution trace is generated as follows.

```
if (verbosity)
  show_in_console(information about the action rule);
```

⇒ Action rules may be: assignments, conditionals, action calling and multicalling.

⇒ Assignments

```
assig_term = data_term;
```

⇒ Conditionals

```
if (conditional_formula) {
  write action rules
} else {
  write action rules
}
```

⇒ Action Calling. If actions belong to components, the calling is preceded by a qualident term.

```
act_name_(param_term,...);
```

⇒ Multicalling rules are implemented by loops. The iterator (*iter*) declared in the loop is a pointer to elements of the set of values from which the range variable is assigned. In each iteration, the value pointed by the iterator is assigned to this variable (*range_vble_name_*).

```
for (set_type::iterator iter = set_term.begin();
```

iter != *set_term*.end();
iter++) {
*set_of_type range_vble_name_* = **iter*;
write action rules
}

⇒ Once all action rules have been executed, if the action has parameters, their values are inserted in the local trace. For each parameter

to_params(pos,*param_name_*);

⇒ Possible behaviour extensions of the action in specialisations and composite objects are called by the function call_extensions.

call_extensions(''*act_name*'',&*param_name_*,...,NULL);

⇒ Possible global behaviour definitions of the action are called by the function call_global_interactions.

call_global_interactions(''*act_name*'',&*param_name_*,...,NULL);

⇒ End of function implementation
} // *act_name*

• check_constraints

This function checks integrity constraints defined in the TROLL class.

⇒ Function head. The function argument indicates whether the object is in its initial state. Its default value is false.

// evaluate constraints
void
class_name::check_constraints(const bool& initial=false) {

⇒ For each constraint definition in the class

⇒ If the constraint needs only to be fulfilled in the initial state, i. e. it was declared *initially*

if (initial) {

⇒ The constraint is checked. Attributes are valuated in the temporary new state. Since not all attributes have been necessarily assigned in the new state, access to attributes are done by means of the function get_*attr_name*_new.

if (verbosity)
show_in_console(*checking constraint constraint_formula*);
if (! *constraint_formula*)
throw_exception(''Constraint not fulfilled: \n *constraint*'');

⇒ If the constraint was declared *initially* close conditional
}

⇒ End function implementation
} // check_constraints

- **get_*attr_name*_new**

This function returns the attribute value in the temporary new state.

⇒ Function head.
// return attribute value in the new state
attr_type class_name::get_*attr_name*_new() {

⇒ If the attribute was assigned in the state transition, the function returns the value of *attr_name*_new, otherwise it returns the value in the previous state *attr_name*_.

```
if (attr_name_new.has_value())
  return attr_name_new;
else
  return attr_name_;
```

⇒ End of function implementation
}

- **init_attributes**

This function initialises attributes declares as *initialised* in the TROLL class.

⇒ Function head
// assign initialised attributes
void
class_name::init_attributes() {

⇒ For each *initialised* attribute

```
if (verbosity)
  show_in_console(''initialising attr_name := const_term'');
attr_name_new = const_term;
```

⇒ End of function implementation
}

• eval_der_attr

This function evaluates derived attributes defined in the specification.

⇒ Function head
// evaluate derived attributes
void
class_name::eval_der_attributes() {

⇒ Each derived attribute is assigned in the temporary new state. As in constraints checking, references to attributes are done by means of the function get_*attr_name*_new.

```
  if (verbosity)
    show_in_console(''assigning derived attribute attr_name := deriv_term'');
  attr_name_new = deriv_term;
```

⇒ End of function implementation
}

• read_attributes

This function read the object attributes from the database.

⇒ Function head
// read attribute values from the database
void
class_name::read_attributes() {

⇒ For each class attribute

⇒ If the attribute has a simple type

```
  read_attr(''attr_name'',''attr_type'',&attr_name);
```

⇒ else the data type is passed in a vector which is built by the function build_type (e. g. build_type(''list'',''int'',NULL))

```
  read_attr(''attr_name'',build_type(type,...,NULL),&attr_name);
```

⇒ If the class is a specialisation, a pointer to the base object is assigned to the attribute superclass.

```
get_superclass(superclass);
```

⇒ End of function implementation

```
}
```

- **write_attributes**

This function stores the object attributes into the database.

⇒ Function head

```
// write attribute values into the database
void
class_name::write_attributes() {
```

⇒ For each class attribute

```
  if (attr_name_new.has_value())
```

⇒ If the attribute has a simple type

```
    write_attr(''attr_name'',''attr_type'',attr_name_new);
```

⇒ else the data type is passed in a vector which is constructed by the function build_type

```
    write_attr(''attr_name'',build_type(type,...,NULL),attr_name_new);
```

⇒ End of conditional

```
  }
```

⇒ End of function implementation

```
}
```

D.1.6 Common Functions

These functions are generated outside the C++ classes. They implement the interface with the animator, the calling of action extensions in composite and specialisation objects, and the global behaviour definition of actions.

- **call_initial_action**

This function represents the interface between the animator and the generated code.

⇒ Arguments of the function are the object class name, the object identity, the action name and the list of input parameters

```
// Call the initial action selected by the user
void
call_initial_action(const string& class_name, identList& id,
                    const string& action_name, const paramList& params) {
```

⇒ For each TROLL class

```
 if (class_name == ''class_name'') {
```

⇒ A variable to hold a pointer to the object is declared.

```
  class_name* obj;
```

⇒ For each non-*hidden* action declared in the class

```
  if (action_name == ''action_name'') {
```

⇒ For each action parameter, a variable is declared.

*param_type param_name_*; ...

⇒ The values of input parameters, that are passed in the function argument params, are assigned to the respective variables.

do_cast(params[*pos*].value, *param_name_*); ...

⇒ The object is loaded on main memory by calling the function get_object defined in the execution manager.

ex_man->get_object (id, &obj, *exec_type*);

⇒ The initial action is called in the object.

obj->*action_name_* (*param_name_*, ...);

⇒ Once the action has been executed, values of output parameters are returned by the function to_out_params which previously checks if they have been assigned. Finally, the function returns.

ex_man->to_out_params(''*param_name*'',''*param_type*'',&*param_name_*);...
return;

⇒ End of conditional checking the action name.

```
  }
```

⇒ End of conditional checking the class name.

```
 }
```

⇒ End of function implementation.

```
}
```

- **call_extensions**

This function implements the calling of action behaviour extensions in specialisation and composite objects.

⇒ Arguments of the function are the class name, the object identity, the action name and a vector of parameter values.

```
// Calling of action behaviour extensions in specialisation and composite objects
void
call_extensions(const string& class_name,identList& id,
                const string& action_name,vector<void*> params){
```

⇒ For each class with specialisations

```
 if (class_name == ''class_name'') {
```

⇒ A pointer to the object is declared and assigned.

```
  class_name* objclass_name;
  ex_man->get_object (id, objclass_name&, NDEF);
```

⇒ For each birth action which causes an object specialisation

```
  if (action_name == ''action_name'') {
```

⇒ For each specialisation aspect caused by the action.

```
   {
```

⇒ If the specialisation is conditioned by an additional formula, input parameters are assigned to local variables and the formula is evaluated. The variable names are those used in the specialisation condition.

```
      param_type param_name_;...
      param_name_ = *(param_type*)params[pos];...
      if (verbosity)
        show_in_console(''checking specialisation condition specialisation_formula'');
      if (specialisation_formula) {
```

⇒ A pointer to the specialisation object is declared and assigned. Additionally, a pointer to the base object is assigned to the attribute superclass of the specialisation object.

```
      if (verbosity)
        show_in_console(''creating specialisation aspect spec_class_name'');
      spec_class_name* objspec_class_name;
      objclass_name->get_spec(objspec_class_name,''spec_class_name'',BIRTH);
      objspec_class_name->get_superclass(objspec_class_name->superclass);
```

⇒ If a behaviour for the birth action has been defined explicitly in the specialisation class, the action is called in the specialisation object.

```
      objspec_class_name->action_name(*(param_type*)params[pos],... );
```

⇒ If no behaviour for the action has been defined in the specialisation object, but the object has *initialised* attributes, the function init_attributes is called in the object.

obj*spec_class_name*->init_attributes();

⇒ If a specialisation formula was defined, end of the conditional.

}

⇒ End of conditional of the specialisation aspect.

}

⇒ End of conditional checking the action name.

}

⇒ For each non-birth action whose behaviour is defined in the specialisation class

if (action_name == ''*action_name*'') {

⇒ For each specialisation class in which the action is defined

⇒ It is checked if the object has the specialisation aspect

if (obj*class_name*->isA(''*spec_class_name*'')) {

⇒ A pointer to the specialisation object is declared and assigned.

*spec_class_name** obj*spec_class_name*;
obj*class_name*->get_spec(obj*spec_class_name*,''*spec_class_name*'',*exec_type*);

⇒ The action is called in the specialisation object

obj*spec_class_name*->*action_name*(*(*param_type**)params[*pos*],...);

⇒ End of the isA conditional

}

⇒ End of conditional checking the action name

}

⇒ End of conditional checking the class name

}

⇒ For each component identification path with actions defined in composite classes

if (ex_man->equal_id_path(id,''*object_name*'',...,NULL)) {

⇒ For each action defined in composite classes

if (action_name == ''*action_name*'') {

⇒ For each composite class in which the action is defined

⇒ For each component identification variable implicitly declared in the action head, a variable is declared. The corresponding identification values, that are ob-

tained from the object identity, are assigned to the new variables.

*param_id_type param_id_name_*; ...
assign_param_id(id,*pos*,*param_id_name_*); ...

⇒ The identity of the composite is built, and a pointer to the composite is declared and assigned.

identList compl_id = get_compl_id(id,*pos*);
*compl_class_name** obj*compl_class_name*;
get_object(compl_id,&obj*compl_class_name*,*exec_type*);

⇒ The action is called in the composite object

obj*compl_class_name*->*action_name*(*param_id_name_*,...,
*(*param_type**)params[*pos*],...);

⇒ End of conditional checking the action name
}

⇒ End of conditional checking the component identification path
}

⇒ End of function implementation
}

• call_global_interactions

This function implements the global behaviour definition of actions.

⇒ As in the previous functions, the arguments of this function are the class name, the object identity, the action name and the vector of parameter values

```
// Implementation of global action behaviour definitions
void
call_global_interactions(const string& class_name,identList& id,
                         const string& action_name,vector<void*> params) {
```

⇒ For each object identification path with global action definitions

```
  if (ex_man->equal_id_path(id,"object_name",...,NULL)) {
```

⇒ For each action with a global behaviour specification

```
    if (action_name == "action_name") {
```

⇒ For each object identification variable implicitly declared in the action head, a variable is declared. The corresponding identification values, that are obtained from the object identity, are assigned to the new variables.

*param_id_type param_id_name_*; ...
assign_param_id(id,*pos*,*param_id_name_*); ...

⇒ For each action parameter, a variable is declared. Parameter values passed in the function argument params, are assigned to the respective variables.

*param_type param_name_*;...
assign_from_param(params[*pos*],*param_name_*);...

⇒ The code associated to each action rule defined in the action behaviour is written. The translation rules are the same as those for the local behaviour definition.

write action rules

⇒ For each output variable, the next function checks if it holds a value and assigns this value to the corresponding entry in the vector of parameters.

assign_to_param(params[*pos*],*param_name_*); ...

⇒ End of conditional checking the action name.
}

⇒ End of conditional checking the object identification path.
}

⇒ End of function implementation.
}

D.1.7 File Structure

The generated code is structured into the following files:

- Each record class is contained in a file whose name consists of the record name suffixed by "_rec.h"
- The C++ class generated from each TROLL class is divided into a header and an implementation file whose names consist of the name of the TROLL class followed by `"_.h"` and `"_.cc"` respectively.
- Common functions are implemented in the file `"execute.cc"`.
- A header file, `troll_classes.h`, contains `"#include"` directives for each class header generated from the TROLL classes. This file is in turn included in the file `"execute.cc"`.
- A `"Makefile"` file contains the list of dependencies between files and all parameters necessary for the compilation. This file is processed by the GNU make utility that automatically calls the C++ compiler and links the code into a library.

D.2 Example

This section shows an extract from the code generated from the CATC example.

• Class Definition of msset

```
class msset_rec {
  public:

    treal press_;
    treal time_;

    bool operator == (const msset_rec& r) const {
      return (press_ == r.press_ && time_ == r.time_);
    }

    bool operator != (const msset_rec& r) const {
      return !(*this == r);
    }

    bool operator < (const msset˙rec& r) const {
      return press_ < r.press_;
    }

    bool has_value() {
      return (press_.has_value() && time_.has_value());
    }

}; // class msset_rec
```

• Class Definition of Application

```
class Application : public troll_class {
  public:

    Application(const identList& ident) : troll_class (ident) {;

    // attributes
    oid<Company> company_;
    oid<Company> company_new;
```

```
    tstring labour_;
    tstring labour_new;
    treal max_pressure_;
    treal max_pressure_new;
    tnat nextexpnr_;
    tnat nextexpnr_new;
    // components
    tmap<tnat,Experiment*> experiments_;

    // methods
    void createappl_(const oid<Company>& comp_,
                     const treal& max_press_,
                     const tstring& lab_);
    void newexperiment_(const tstring& nam_,
                        const msset_rec& st_,
                        tnat& expnr_);
    void deleteappl_();

    // check constraints
    void check_constraints (const bool& initial=false);
    // return attribute value in the new state
    oid<Company> get_company_new();
    tstring get_labour_new();
    treal get_max_pressure_new();
    tnat get_nextexpnr˙new();
    // assign initialised attributes
    void init_attributes();
    // read attribute values from the database
    void read_attributes();
    // write attribute values into the database
    void write_attributes();

}; // class Application
```

• Class Implementation of Application

```
void
Application::createappl_(const oid<Company>& comp_,
                         const tstring& lab_,
                         const treal& max_press_) {
  // Control Code
  vector<local_trace>::iterator i;
  for(i=trace.begin(); i != trace.end(); i++)
```

```
    if(i->action=="createAppl")
      if(equal(i->params[0],comp_) &&
         equal(i->params[1],lab_) &&
         equal(i->params[2],max_press_))
          return;

  check_birth("createAppl");
  int pos = to_trace("createAppl",&comp_,&lab_,&max_press_,NULL);
  init_attributes();

  // Behaviour code
  if (verbosity)
    show_in_console("assigning company := comp");
  company_new = comp_;
  if (verbosity)
    show_in_console("assigning max_pressure := max_press");
  max_pressure_new = max_press_;
  if (ex_man->verbosity)
    ex_man->show_in_console("assigning labour := lab");
  labour_new = lab_;

  // Control Code
  to_params(pos,comp_);
  to_params(pos,lab_);
  to_params(pos,max_press_);
  call_extensions("createAppl",&comp_,&lab_,&max_press_,NULL);
  call_global_interactions("createAppl",&comp_,&lab_,&max_press_,NULL);
}


void
Application::newexperiment_(const tstring& nam_,
                            const msset_rec& st_,
                            tnat& expnr_) {
  // Control Code
  vector<local_trace>::iterator i;
  for(i=trace.begin(); i != trace.end(); i++)
    if(i->action=="newExperiment")
      if(equal(i->params[0],nam_) &&
         equal(i->params[1],st_)) {
          assign_param(i->params[2],expnr_);
          return;
      }

  int pos = to_trace("newExperiment",&nam_,&st_,&expnr_,NULL);
```

```
  // Behaviour code
  // Check precondition
  if(!(st_.press_ <= max_pressure_))
    throw_exception("Precondition not fulfilled:\n\n  onlyIf(st.press <= max_pressure)");

  if (verbosity)
    show_in_console("calling Experiments(nextExpNr).createExp(nam,st)");
  get_comp(experiments_,nextexpnr_,"Experiment","Experiments","tnat",BIRTH);
  experiments_[nextexpnr_]->createexp_(nam_,st_);
  if (verbosity)
    show_in_console("assigning expNr := nextExpNr");
  expnr_ = nextexpnr_;
  if (verbosity)
    show_in_console("assigning nextExpNr := nextExpNr+1");
  nextexpnr_new = nextexpnr_+1;

  // Control Code
  to_params(pos,nam_);
  to_params(pos,st_);
  to_params(pos,expnr_);
  call_extensions("newExperiment",&nam_,&st_,&expnr_,NULL);
  call_global_interactions("newExperiment",&nam_,&st_,&expnr_,NULL);
}

void
Application::deleteappl_() {
  // Control Code
  if (in_trace("deleteAppl"))
    return;

  check_death("deleteAppl");
  to_trace("deleteAppl");

  // This action has not an explicit behaviour

  // Control Code
  call_extensions("deleteAppl");
  call_global_interactions("deleteAppl");
}

// evaluate constraints
void
Application::check_constraints(const bool& initial=false) {
  if (verbosity)
```

```
    show_in_console(``checking constraint nextExpNr <= 11''');
  if(!(nextexpnr_new <= 11))
    throw_exception("Constraint not fulfilled:\n   nextExpNr <= 11");
}

// assign initialised attributes
void
Application::init_attributes() {
  if (verbosity)
    show_in_console(``initialising nextexpnr := 1'');
  nextexpnr_new = 1;
}

// read attribute values from the database
void
Application::read_attributes() {
  read_attr("company",build_type ("OID","Company",NULL),&company_);
  read_attr("labour","string",&labour_);
  read_attr("max_pressure","real",&max_pressure_);
  read_attr("nextExpNr","nat",&nextexpnr_);
}

// write attribute values into the database
void
Application::write_attributes() {
  if (company_new.has_value())
    write_attr("company",build_type ("OID","Company",NULL),company_new);
  if (labour_new.has_value())
    write_attr("labour","string",labour_new);
  if (max_pressure_new.has_value())
    write_attr("max_pressure","real",max_pressure_new);
  if (nextexpnr_new.has_value())
    write_attr("nextExpNr","nat",nextexpnr_new);
}
```

- **call_initial_action**

```
// Call the initial action selected by the user
void
call_initial_action(const string& class_name, identList& id,
                    const string& action_name, const paramList& params) {
  if (class_name == "Application") {
    Application* obj;
```

```
    if (action_name == "createAppl") {
      oid<Company> comp_;
      tstring lab_;
      treal max_press_;

      ex_man->do_oid_cast(params[0].values, comp_);
      do_cast(params[1].values, lab_);
      do_cast(params[2].values, max_press_);

      ex_man->get_object (id, &obj, BIRTH);
      obj->createappl_ (comp_,lab_,max_press_);
      return;
    }
    if (action_name == "newExperiment") {
      tstring nam_;
      msset_rec st_;
      tnat expnr_;

      do_cast(params[0].values, nam_);
      do_record_cast(params[1].values, st_);

      ex_man->get_object (id, &obj, UPD);
      obj->newexperiment_ (nam_,st_,expnr_);
      ex_man->to_out_params("expNr","nat",&expnr_);
      return;
    }
    if (action_name == "deleteAppl") {
      ex_man->get_object (id, &obj, DEATH);
      obj->deleteappl_ ();
      return;
    }
    return;
  }
  ...
}
```

- **call_extensions**

```
// Calling of action behaviour extensions in specialisation and composite objects
void
call_extensions(const string& class_name, identList& id,
                const string& action_name, vector<void*> params) {
```

```
  if (class_name == "User") {
    User* objUser;
    ex_man->get_object (id, &objUser, NDEF);
    if (action_name == "login") {
      {tstring n_;
        n_=*(tstring*)params[0];
        tstring d_;
        d_=*(tstring*)params[1];
        tstring t_;
        t_=*(tstring*)params[2];
        if (verbosity)
          show_in_console(``checking specialisation condition of Staff: t = \"staff\"'');
        if(t_ == "staff") {
          if (verbosity)
            show_in_console(``creating specialisation aspect Staff'');
          Staff* objStaff;
          objUser->get_spec(objStaff,"Staff",BIRTH);
          objStaff->get_superclass(objStaff->superclass);
        }
      }
      {tstring n_;
        n_=*(tstring*)params[0];
        tstring d_;
        d_=*(tstring*)params[1];
        tstring t_;
        t_=*(tstring*)params[2];
        if (verbosity)
          show_in_console(``checking specialisation condition of Operator t = \"operator\"'');
        if(t_ == "operator") {
          if (verbosity)
            show_in_console(``creating specialisation aspect Operator'');
          Operator* objOperator;
          objUser->get_spec(objOperator,"Operator",BIRTH);
          objOperator->get_superclass(objOperator->superclass);
        }
      }
    }
  }
}
```

- **call_global_interactions**

```
// Implementation of global action behaviour definitions
```

```
void
call_global_interactions(const string& class_name,identList& id,
                         const string& action_name,vector<void*> params) {
  if (ex_man->equal_id_path(id,"Users","Staff",NULL)) {
    if (action_name == "createExperiment") {
      tnat userid_;
      assign_param_id(id,0,userid_);

      tnat appnr_;
      assign_from_param(params[0],appnr_);
      tstring nam_;
      assign_from_param(params[1],nam_);
      msset_rec st_;
      assign_from_param(params[2],st_);
      tnat expnr_;
      assign_from_param(params[3],expnr_);

      // Behaviour code

      if (verbosity)
        show_in_console("calling IG34.Applications(appNr).newExperiment(nam,st,expNr)");
      Group* obj;
      ex_man->get_global_obj(obj,"Group","IG34",UPD);
      obj->get_comp(obj->applications_,appnr_,"Application","Applications","tnat",UPD);
      obj->applications_[appnr_]->newexperiment_(nam_,st_,expnr_);

      assign_to_param(params[3],expnr_);
    }
    ...
  }
  ...
}
```